NORTHWESTERN
UNIVERSITY

Computer Science Department

Technical Report
NWU-CS-03-16
November 24, 2003

Tsunami: A Wavelet Toolkit for Distributed Systems

Jason A. Skicewicz   Peter A. Dinda

Abstract

This paper describes the design and implementation of Tsunami, a wavelet-based library built to encompass the range of research from offline analysis of computer generated resource signals to the construction and deployment of online systems. Wavelet analysis has proven to be an invaluable analysis technique for discovering characteristics of signals and has been applied to many areas related to computer systems research. Tsunami is created mostly for use in distributed systems, domains where sensors are deployed to sample resource signals related to hosts and networks, and are used for making run-time decisions in applications. From the analysis of computer generated resource signals, online systems may be deployed using our toolkit to provide performance gains in user applications. With Tsunami, a user can seamlessly transition from simulation to deployment of a wavelet-based online system. The toolkit design is extremely general in that the provided interfaces can be used for almost any type of application that may benefit from wavelet approaches. It is also extensible and flexible, allowing users to customize their analysis using the coarse- and fine-grain building blocks provided in the toolkit. In this paper, we summarize the techniques of wavelet analysis for use in computer systems and provide implementation details of the library to provide a user with the power to wavelet-enable their application. We describe how the toolkit can be extended, how it performs in terms of sample rates and scalability, and how it can be used with the RPS toolkit to build distributed wavelet systems. Conclusions and future directions of our research related to this toolkit will be discussed. Tsunami is available from http://www.cs.northwestern.edu/~RPS.

Effort sponsored by the National Science Foundation under Grants ANI-0093221, ACI-0112891, ANI-0301108, EIA-0130869, and EIA-0224449. Any opinions, findings and conclusions or recommendations expressed in this material are those of the author and do not necessarily reflect the views of the National Science Foundation (NSF).

Keywords: wavelet analysis, signal processing, distributed systems, resource monitoring

Tsunami: A Wavelet Toolkit for Distributed Systems

Jason A. Skicewicz Peter A. Dinda
{jskitz, pdinda}@cs.northwestern.edu
Department of Computer Science
Northwestern University

November 24, 2003

Abstract

This paper describes the design and implementation of Tsunami, a wavelet-based library built to encompass the range of research from offline analysis of computer generated resource signals to the construction and deployment of online systems. Wavelet analysis has proven to be an invaluable analysis technique for discovering characteristics of signals and has been applied to many areas related to computer systems research. Tsunami is created mostly for use in distributed systems, a domain where sensors are deployed to sample resource signals related to hosts and networks, and are used for making run-time decisions in applications. From the analysis of these computer generated resource signals, online systems may be deployed using our toolkit to provide performance gains in user applications. With Tsunami, a user can seamlessly transition from simulation to deployment of a wavelet-based online system. The toolkit design is extremely general in that the provided interfaces can be used for almost any type of application that may beenefit from wavelet approaches. It is also extensible and flexible, allowing users to customize their analysis using the coarse- and fine-grain building blocks provided in the toolkit. In this paper, we summarize the techniques of wavelet analysis for use in computer systems and provide implementation details of the library to provide a user with the power to wavelet-enable their application. We describe how the toolkit can be extended, how it performs in terms of sample rates and scalability, and how it can be used with the RPS toolkit to build distributed wavelet systems. Conclusions and future directions of our research related to this toolkit will be discussed.

Tsunami can be downloaded from http://www.cs.northwestern.edu/~RPS.

Effort sponsored by the National Science Foundation under Grants ANI-0093221, ACI-0112891, ANI-0301108, EIA-0130869, and EIA-0224449. Any opinions, findings and conclusions or recommendations expressed in this material are those of the author and do not necessarily reflect the views of the National Science Foundation (NSF).

1 Introduction

Distributed systems are becoming increasingly adaptive, and scheduling and other adaptation decisions are being made at application, middleware, and OS levels. These decisions are made according to the availability and load of machines, and also the network conditions that prevail between the machines. Resource monitoring systems provide this information, typically in the form of appropriately sampled [1] periodic measurements, or discrete-time *resource signals*.

There are many uses of resource signals and operations than can be applied to them, but the following are, in our view, most significantly impacted by wavelet analysis:

- *Characterization*. Characterizing the dynamic behavior of classes of signals provides us with insights into the behavior of resources, their schedulers, and their human users.
- *Summarization*. Summarizing one or a collection of signals makes it easier for a human or a software system to answer questions.
- *Dissemination*. Conveying a resource signal or collection of resource signals from sensors to a collection of users should be done with as little network traffic as possible.
- *Prediction*. Adaptive systems care about the future behavior of the resource signal.

To further our (and others') research along these and other directions, we have developed Tsunami, a wavelet toolkit for use with resource signals in distributed systems. Tsunami's design is general, and extentsible, making it useful in other domains as well.

Why wavelet analysis? While there are many analysis techiniques applicable to resource signals, most have difficulties when faced with non-stationary or non-periodic behavior. Wavelet techniques overcome these deficiencies, and have been shown to be a powerful analysis tool for understanding signals in general and, more specifically, computer resource signals such as host load [21] and network bandwidth [16]. A wavelet transform converts a periodically sampled, time-domain signal into two dimensions representing time and scale (like frequency). The outputs of the wavelet transform are called the *wavelet coefficients*, and can be studied in lieu of the original signal for they contain all the information in it. Often a more flexible analysis of the signal, called a *multi-resolution analysis (MRA)*, is preferred. Characterization, summarization, dissemination, and prediction in wavelet domain exposes opportunities that do not exist in time-domain or frequency-domain. A wavelet domain signal can be readily converted back to time-domain without loss of information.

Why Tsunami? Many implementations of wavelet analysis exist. However, none of them is tuned for use in resource monitoring in a distributed system, where we require inexpensive streaming operation at low sample rates and efficient communication over lossy channels. In addition, few provide generality, allowing the user to construct essentially arbitrary transforms in pursuit of research goals. The Tsunami toolkit can build arbitrary transforms and can adaptively shape its transforms at run-time. None that we have found, can be readily used for both offline and online analysis. In addition, it is difficult to find efficient wavelet tools that are not just built for use on digital signal processors and many tools that will run on PCs are built for offline analysis using

[1] To the best of our knowledge, no one has studied the problem of determining an appropriate rate of sampling for resources in distributed computing. This is in our future plans.

scripting languages and are therefore not efficient for use in deploying online systems. Finally, the toolkit's design is optimized to fit nicely into the RPS system [5] for communication, prediction and monitoring.

The Tsunami toolkit is built using the C++ programming languange and the Standard Template Library (STL) generic container types. Tsunami can run with many different input data types, and can be easily extended for use in systems research and system building.

The remainder of the paper is as follows. In Section 2, we begin by discussing related research and differentiating Tsunami from other wavelet-based systems. Section 3 describes the deficiencies of other analysis techniques, and details the mathematical foundations of the wavelet transform and the theory of multi-resolution analysis. Section 4 lays out the goals and requirements of Tsunami, working from the perspective of a researcher using our system.

Of particular interest to a researcher already familiar with wavelets, Section 5 describes how to get started using Tsunami: downloading, compiling, using the command-line tools, and writing new tools. Section 6 continues by describing the design and implementation of the toolkit, while Section 7, describes advanced uses of Tsunami and how to extend it.

Performance is a critical goal in a system designed for use in online environemnts, such as Tsunami. In Section 8, we measure the performance of the Tsunami toolkit in terms of how the system scales and performs at high sampling rates. We also discuss real-time system delays, a property that is fundamental to wavelet systems based on causal filters.

We generally use Tsunami along with the Resource Prediction System (RPS) toolkit [5]. RPS provides communication services, prediction services, sensors, and many other tools. Section 9 details the interface of Tsunami to RPS. Using the interface, we have built a number of RPS components that include wavelets. Using the combination, we have explored using the predictive models that RPS offers on the wavelet coefficients as an approach to reducing real-time system delay. We have also built a proof-of-concept resource signal dissemination and query system in which applications can subscribe to a stream of samples at a rate and resolution that is right for their application. Finally, in Section 10 we provide concluding remarks and the roadmap of our future work.

2 Related work

We are not the first to see the usefulness of using wavelet-based techniques for studying the behavior of signals and building online systems. Researchers have applied wavelet-based techniques to understand network traffic and packet traces for some time. The self-similar nature of network traffic was an important discovery in the early 90s. Abry, et al, have developed wavelet-based techniques to estimate the Hurst parameter, the degree of self-similarity [1]. Feldmann, et al have extensively used wavelets to characterize network traffic as multi-fractal [6] and to study the impact of this property on control mechanisms such as TCP congestion control [7]. Riedi, et al have shown how to use wavelets to synthesize network traffic [17], computing results in an efficient manner that appear to match real Ethernet traces visually and statistically. Qiao, et al have empirically studied the *predictability* of wavelet coefficients of real network traffic for use in determing message transfer times [16]. Several wavelet systems also exist. For example, WIND uses wavelet-based scaling analysis to detect network performance problems [9]. Another relevant system estimates the Hurst parameter at the router to make adaptive changes in congestion control, or to provide up to date information about traffic dynamics without storing all of the data for offline analysis [18]. We have proposed another online wavelet system for the dissemination of resource information in an accurate and scalable fashion to applications of various granularity [21], a goal achieved through use of the Tsunami toolkit. This proposal is discussed further in Section 9.

In this report, we detail Tsunami, a wavelet library that we have constructed to help better understand the dynamics of computer generated resource signals. From this, we hope to create and deploy distributed, online components for applications that can benefit from having wavelet enabled applications for use in scheduling and other domains. The library has been built to satisfy many of our research needs that have not been met elsewhere. There are many offline wavelet analysis tools available that provide wavelet decompositions and transformations related to wavelets. Tools that we have used include the Wavelet Toolbox in Matlab [24] and the Matlab scripts created by D. E. Newland in his book on spectral analysis [15]. However, many of the goals of our research are of a more dynamic, more adaptable, and general nature necessitating the enhanced functionality and flexibility over existing tools.

In order to address generality, we have built the tool with fine-grain building blocks in order to create any type of decomposition, not limited simply to the structure of the wavelet transform and wavelet packets. The toolkit is composed of filters (FIR, IIR, etc.), coefficients for filters, downsamplers and upsamplers parameterized by the rate, and stages that aggregate the fine-grain blocks into two-band structures. From the building blocks, we create many standard wavelet transformations, and have included interfaces for multi-resolution analysis (MRA) for obtaining the *approximation* and *detail* signals. MRA will be discussed in more detail in Section 3. In addition, we provide interfaces for getting combinations of these signals including the well known transform, consisting of one coarse approximation signal and a set of detail signals, and any other possible mix of MRA signals. This arms the users of our toolkit with the power to look at many different types of signals that may be useful to their research, not limiting the user to standard transforms. This has proven invaluable in our study of the predictability of network bandwidth, where we have shown that binning and MRA analysis on approximation signals are similar [16].

Many of the existing offline tools are not as run-time friendly as we think necessary to adapt to quick changing resource signals. In the Tsunami toolkit, we provide interfaces to dynamically adapt the transform and the decomposition to the characteristics of the input resource signal. We find that the static transformations that are typically found in offline toolboxes are insufficient to properly study how to adapt the analysis to the changes in a given input signal. We envision a user of the toolbox dynamically adding or removing levels in the decomposition based on epoch changes detected in the resource signal. At this time, the toolbox does not contain mechanisms for detecting epoch changes, but is something that we plan to look into in our future research. In addition to dynamic structural changes, the toolkit allows for dynamically changing the wavelet basis functions at run-time. This can lessen computational complexity of the system when a lower order basis function can provide as much benefit as a higher order one. This typically occurs when a particular frequency band has low energy and therefore the filtering of this band is not showing anything interesting. The idea of time-varying operation, adapting parameters of the transform to the signal characteristics at run-time, is a very appealing area of research that we hope may lead to performance gains in prediction and scheduling in distributed systems. Many of the dynamic interfaces that have been created in Tsunami is informed by the work of Sodagar et al [23].

The Tsunami library can also be used to build online, wavelet-enabled systems. One application that we have thought about in detail is in resource dissemination and resource prediction in distributed applications. However, the library can be used for building other types of online systems as well. An extremely powerful benefit of the toolkit, is that the transition from analysis/simulation, to building online systems is virtually effortless. The library can be used in either mindset right out of the box. This is a powerful advantage over Matlab even though Matlab contains utilities for compiling ”m-file” scripts into executables.

Additionally, if a user wants to use Tsunami in conjunction with the Resource Prediction System (RPS), the Tsunami toolkit will be included with the next RPS version release. The RPS release code contains interface classes between Tsunami and RPS so that researchers can extend current applications to use wavelets with time-series analysis and RPS communication constructs.

3 Wavelet discussion

In order to motivate the reasons for why wavelets are a powerful tool for signal analysis, it is informative to highlight the deficiencies of other analysis techniques. The Fast Fourier Transform (FFT), a technique used for analyzing the frequency content of a signal, only provides frequency information, showing nothing about how a signal changes in time. Due to this limitation, the time-domain representation cannot be exactly reconstructed from the FFT output unless the input is periodic in time. The Short-term Fourier Transform (STFT) attempts to solve this limitation by analyzing signal changes in both time and in frequency. The STFT performs an FFT over a fixed size window that slides over time. The time dynamics are captured by viewing the signal in a fixed size window, and the frequency by the computation of the FFT over that window. This technique provides an estimate of how the signal changes in both time and frequency, but is limited by the window size. If the window size is made small, the analysis provides fine-grain time resolution and coarse-grain frequency resolution. As the window size increases, the converse is true. Therefore it is said that the STFT suffers from the Heisenberg Uncertainty Principle between exact knowledge of either time or frequency. The wavelet analysis overcomes both limitations of the FFT and the STFT, and is therefore an extremely useful technique for analyzing signals that change rapidly in both time and frequency, a characteristic that computer generated signals tend to exhibit.

In Figure 1, we describe the general symbols used in this section to describe the signal processing aspects of wavelet analysis. In Figure 2, we describe symbols specific to the MRA wavelet analysis.

In what follows, we provide an overview of wavelets as a tool for analysis, and what it means to provide a multi-resolution view of a signal. In our work, we take one-dimensional resource signals, periodically sampled, and trasform them into two-dimensions, representing time and scale (like frequency). The signal is decomposed into a number of *levels* representing not only bands of frequency information, but also the time dynamics that occur in each frequency band. The wavelet technique is the superior method to view how a signal changes in a frequency band at a particular time, and is said to exhibit good *time-frequency locality*.

As an example of a wavelet decomposition, in Figure 3 (a), we show a trace of the load on a host machine spanning 8192 seconds (approximately $2\ 1/2$ hours) sampled at a 1Hz rate. The host load signal is then input into a wavelet transform block, thus decomposing the signal into

Symbol	Description
General signal processing	
ΔT	Interval of time between subsequent samples.
n	Represents the short form of nT, but since the signals that we encounter in our work are periodically sampled, T is implied.
f	Frequency in Hz and $f = 1/T$.
f_s	The sampling frequency in Hz and $f_s = 1/\Delta T$.
ω	Angular frequency in radians/sec and $\omega = 2\pi f$
ω_s	The sampling angular frequency in radians/sec and $\omega_s = 2\pi f_s$.
$x(n)$ or x_n	Input resource signal indexed by n.
M	The number of two-band filter banks used to decompose a signal. It is also the number of individual approximation and detail signals available.
$g_a(n)$	The filter coefficients of the low-pass analysis filter.
$G_a(z)$	The Z-transform of the low-pass analysis filter (frequency response).
$h_a(n)$	The filter coefficients of the high-pass analysis filter.
$H_a(z)$	The Z-transform of the high-pass analysis filter (frequency response).
$y_i(n)$	Subband signals that are output from the process of filtering and down sampling.
$g_s(n)$	The filter coefficients of the low-pass synthesis filter.
$G_s(z)$	The Z-transform of the low-pass synthesis filter (frequency response).
$h_s(n)$	The filter coefficients of high-pass synthesis filter.
$H_s(z)$	The Z-transform of the high-pass synthesis filter (frequency response).
c	A constant scaling factor to compare x_n with $\hat{x}_n$.
$\hat{x}(n)$ or $\hat{x}_n$	The reconstructed signal from the subband signals $y_i(n)$.
n_d	Integer-valued system delay. Value of delay is dependent on structure and filter order.

Figure 1: Table of general signal processing symbols used to describe the mathematical foundation of wavelet analysis.

14 levels yielding the wavelet coefficients. In (b), the squared wavelet coefficients are shown in three dimensionions. The brightness of each block then corresponds to the energy of each wavelet coefficient. The y-axis corresponds to the scale of the transform, and the x-axis corresponds to time. Peering into this picture qualatatively, the impulses in time that represent a sharp increase in system load are seen. As an example, the impulse that occurs around sample 4000 in (a) is shown at many scales in the energy plot of the wavelet coefficients. Each level captures the same amount of time, but each level has fewer coefficients by a factor of two. To make this point more concrete, in Figure 4 we show by level number the period between coefficients (ΔT), the number of samples in each scale, and the frequency content that is captured in each band if we assume that the input signal is band-limited to a frequency of $f_s/2$. This figure is directly matched to the information in Figure 3.

As we learn more about wavelets, the information described in the previous paragraph will become more clear. Next we describe the basic building blocks of a wavelet analysis.

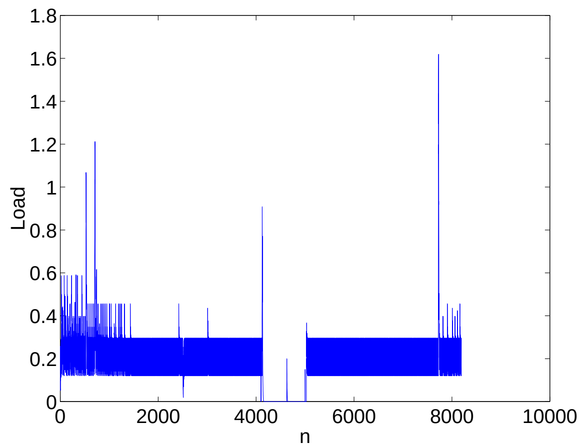
Symbol	Description
Multi-resolution analysis	
$approx_j$	The j^{th} approximation signal in the decomposition.
$detail_j$	The j^{th} detail signal in the decomposition.
j	Subspace indice that ranges from $0, \ldots, M$, and represents which scale.
n	Sample indice that ranges from $0, \ldots, 2^j$ and represents which coefficient.
ψ_0	The mother wavelet function.
ϕ_0	The scaling function.
$\psi_{j,n}$	The band-pass wavelet function at scale j, coefficient n.
$\phi_{j,n}$	The low-pass scaling function at scale j, coefficient n.
t	Represents time in seconds.
$\{V_j\}$	A collection of nested subspaces.
$a_x(j,n)$	The approximation coefficients produced from input signal x at scale j and coefficient n.
$d_x(j,n)$	The detail coefficients produced from input signal x at scale j and coefficient n.
$Proj_X Y$	The projection of signal X into subspace Y.

Figure 2: Table of multi-resolution analysis symbols used to describe the mathematical foundation of wavelet analysis.

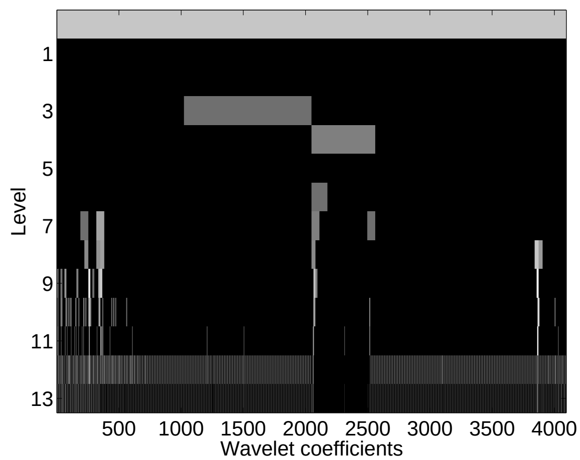
3.1 The basic building blocks and structures

To start our discussion, we will detail the building blocks of the wavelet transform, the two-channel digital filter bank shown in Figure 5. From this, all other representations from uniform filter banks to non-uniform filter banks and the wavelet decomposition follow naturally. In the most simple case, an input resource signal, $x(n)$, a bandlimited signal with typical input spectrum shown in (a), periodically sampled at a rate of $\omega_s = 2\pi$, is decomposed into two bands using digital filters. As shown in (b), $x(n)$ is input into the filters $G_a(z)$, a low pass filter, and $H_a(z)$, a high pass filter. The magnitude response of these two filters are shown in (c). The purpose of these two filters is to split the underlying resource signal into two half band signals representing the high and low frequency information. Since the low and high frequency information now only contain half the information as before the operation, each output can be resampled down by a factor of two without loss of information. The operation of resampling down by a factor of two is known as decimation. The outputs of the filter are typically called the *subband signals*. These signals, designated by $y_0(n)$ and $y_1(n)$, are resampled to half the original sample rate, and represent two orthogonal slices of frequency in the input resource signal.

The subband signals can be manipulated in some way based on the application. For example, a compression application might look for the band with less tonal information, and reduce the amount of information in this band appropriately. Once an application does the appropriate manipulation of the subband signals, it is then appropriate to expand the signal by two to obtain the original sampling rate, and then reconstruct the resource signal using the synthesis filters $G_s(z)$ and $H_s(z)$. If the analysis and synthesis filters are designed accordingly, the reconstructed signal, $\hat{x}(n)$, can be exactly the input signal neglecting quantization noise and delay due to causal filters. The system is said to have the perfect reconstruction (PR) property if $x_(n) = c \cdot \hat{x}(n - n_d)$ for



(a)



(b)

Figure 3: Example wavelet decomposition for resource signal host load: (a) host load trace for 8192 seconds, (b) mean square energy of the wavelet coefficients (black indicates low energy, light indicates higher energy).

some $c \neq 0$ and some integer n_d [25]. Much work has gone into designing the properties of perfect reconstruction for structures of this type. Most notably is the method known as the conjugate quadrature filter (CQF) method by Smith and Barnwell in 1984 and later published as a journal article in 1986 [22]. We use the CQF method in the Tsunami toolkit to derive from the filter $G_a(z)$, the coefficients for the other three filters, $H_a(z)$, $G_s(z)$ and $H_s(z)$.

From the general structure of the two-band filter bank, many other signal decompositions can be constructed. The most notable decomposition represented in our toolkit is that of the tree-structured wavelet decomposition. This is shown in Figure 6. In (a), the structure is shown as a tree whose nodes grow in one direction, where the lowest frequency component of the two-band split is input into another two-band filter bank. This continues up the tree until the signal has been decomposed into the proper number of bands, designated by $M + 1$. In many systems, the multiscale representation is manipulated in some clever way or sent over the network to be reconstructed by various distributed applications. The reconstruction part of the structure has the

Decomposition level	Period in seconds	Number of points	Frequency low	Frequency high
Input = 1Hz	1	n	0	$f_s/2$
0	2	$n/2$	$f_s/4$	$f_s/2$
1	4	$n/4$	$f_s/8$	$f_s/4$
2	8	$n/8$	$f_s/16$	$f_s/8$
3	16	$n/16$	$f_s/32$	$f_s/16$
4	32	$n/32$	$f_s/64$	$f_s/32$
5	64	$n/64$	$f_s/128$	$f_s/64$
6	128	$n/128$	$f_s/256$	$f_s/128$
7	256	$n/256$	$f_s/512$	$f_s/256$
8	512	$n/512$	$f_s/1024$	$f_s/512$
9	1024	$n/1024$	$f_s/2048$	$f_s/1024$
10	2048	$n/2048$	$f_s/4096$	$f_s/2048$
11	4096	$n/4096$	$f_s/8192$	$f_s/4096$
12	8192	$n/8192$	$f_s/16384$	$f_s/8192$
13	8192	$n/8192$	0	$f_s/16384$

Figure 4: Following the diagram of Figure 3 (b), we show the decomposition level, the period (ΔT), the number of points at each level ($n =$ number of points at 1Hz rate) and the frequency range information at each level in the decomposition.

reverse tree with pairs of synthesis filter banks matched up with the various levels of the analysis filter banks. This tree structured filter bank has a non-uniform decomposition of the input resource signal and is shown (not exactly to scale) in (b). An equivalent structure to that represented in (a) is shown in Figure 7 (a). In this figure all the various filters of the tree have been convolved into a single filter followed by a single down sampling component of the appropriate rate. Since the operation is dyadic, non-uniform, and logarithmically decomposed, the downsampling rates increase from bottom to top by powers of two. In (b), the general structure of filter banks are shown, representing both non-uniform and uniform decompositions. In a uniform decomposition, each of the r_i are equivalent to the number of bands in the decomposition. For example, if we use a 10 level decomposition, $r_i = 10$ for all i.

There are numerous tradeoffs between the structures, and the types of filters that are used in the analysis and synthesis stages. A major tradeoff is the system delay between the input resource signal, $x(n)$, and the reconstructed system output, $c \cdot \hat{x}(n-n_d)$. Because in all cases it is beneficial to minimize the real-time system delay, the type of structure and the order of filter coefficients must be chosen carefully. Later in this report, Section 8, we provide an analytical analysis of the real-time system delay based on the non-uniform structure, the operation type and the order of the filter coefficients. Real-time system delay is an important issue that must be addressed to realize interactive, wavelet-based distributed system application building. It is an important design consideration that is typically based on the delay signature of the application.

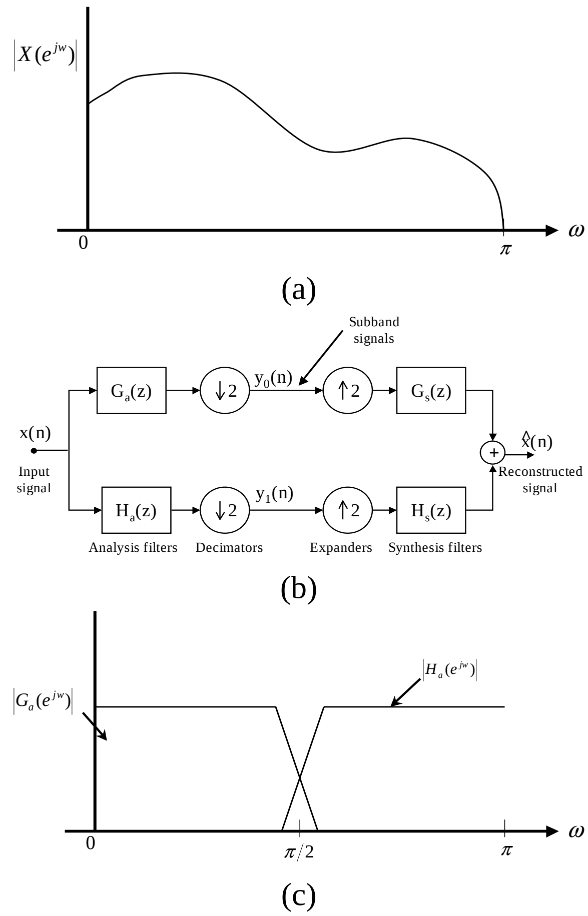


Figure 5: Simple discussion of wavelets: (a) typical spectrum of resource signal, (b) the two-channel digital filter bank, (c) typical filter responses of two-band filter bank.

3.2 Multi-resolution analysis

It is important to describe to the reader what is meant by a wavelet-based multi-resolution analysis. Figure 8 shows this qualitatively. The figure shows an input signal x_n, representing an appropriately sampled, fine-grain resource signal, the binning of a network bandwidth trace. The input signal is being decomposed into three resolution stages composed of *approximation* and *detail* signals. By traversing the approximation tree ($approx_j$, j increasing), we observe that each of the plots not only have fewer points, but describe a coarser approximation of the underlying input signal. Each successive approximation contains half the number of points and captures half of the frequency content of the previous approximation. Even though as j increases each approximation has fewer points, each graph is still covering the same period of time. By observing the details, we can qualitatively see that the amount of information taken away from each approximation at subsequent levels is the detail. That is, $approx_j = approx_{j-1} - detail_j$. The filters ψ and ϕ are derived from the wavelet basis function.

The following discussion is informed by the work of Mallat [12], Daubechies [4], and Abry, et al [2]. The structure shown in the figure is the discrete wavelet transform (DWT), a mathematical transformation for representing a 1-dimensional discrete time signal x_n. Intuitively, the DWT splits

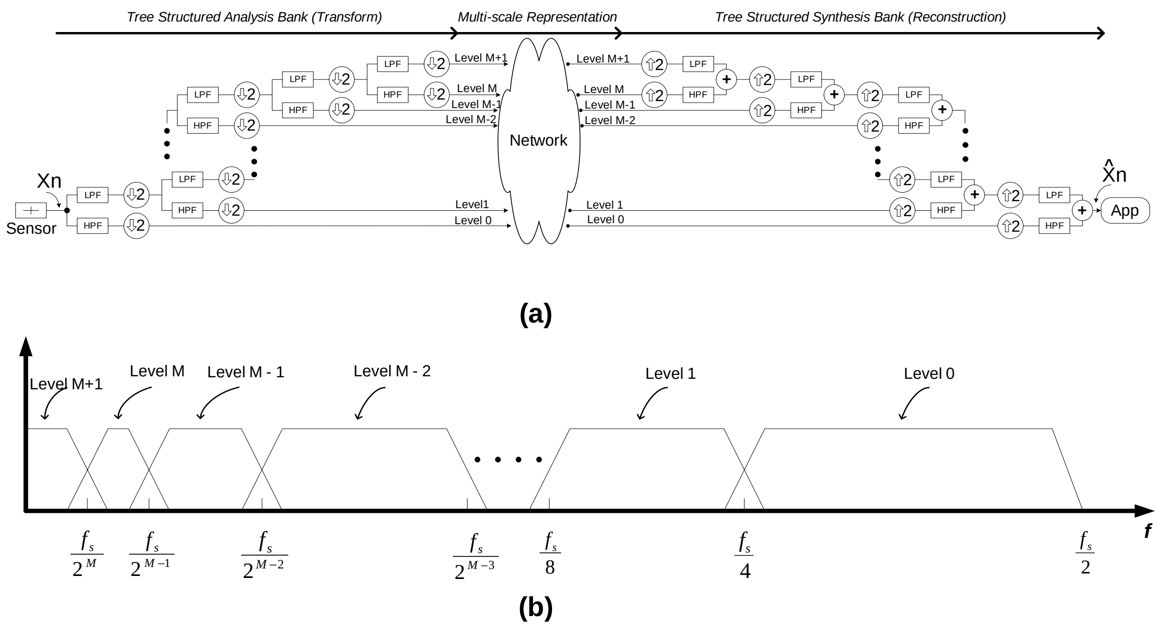


Figure 6: Tree structure filter bank system: (a) diagram of the system, (b) frequency representation.

a 1-dimensional signal into a 2-dimensional signal representing time and scale (like frequency) information. The input signal is represented in terms of shifted and dilated versions of a prototype bandpass wavelet function $\psi_{j,n}$ and shifted versions of a low pass scaling function $\phi_{j,n}$, based on the scaling function, ϕ_0 and the mother wavelet basis function, ψ_0. The relationship between these functions are

$$\{\phi_{j,n}(t) = 2^{-j/2}\phi_0(2^{-j}t - n), n \in \mathcal{Z}\}$$

and

$$\{\psi_{j,n}(t) = 2^{-j/2}\psi_0(2^{-j}t - n), n \in \mathcal{Z}\}.$$

To generate an accurate multi-resolution view of the input signal, the functions ψ_0 and ϕ_0 are chosen so that they are of sufficiently high order (typically determined empirically) and constitute an unconditional Riesz basis. More details on the properties of the wavelet and scaling functions can be found in Daubechies [4] and Newland [15, Chapter 17]. Multi-resolution analysis (MRA) first coined by Mallat [12], consists of a collection of nested subspaces $\{V_j\}_{j\in Z}$ such that:

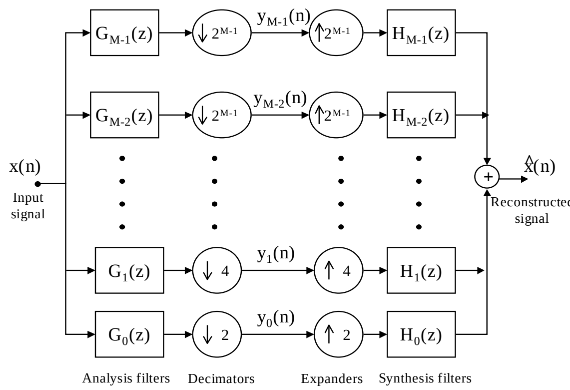
$$V_j \subset V_{j-1}.$$

Multi-Resolution analysis projects the signal x_n into each of the approximation subspaces V_j. The approximation signal is then given by the following relationship:

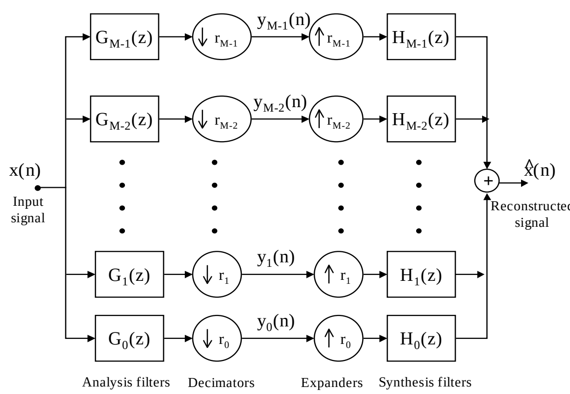
$$approx_j(t) = (Proj_{V_j} x_n)(t) = \sum_n a_x(j,n)\phi_{j,n}(t).$$

The coefficients $a_x(j,k)$ are defined through the inner product of the input signal x_n with $\phi_{j,n}$,

$$a_x(j,n) = \langle x_n, \phi_{j,n}\rangle.$$



(a)



(b)

Figure 7: Filter bank structures: (a) the equivalent non-uniform structure of the tree, (b) the general filter bank structure.

Similarly, the detail signal is given by the following relationship:

$$detail_j(t) = (Proj_{W_j} x_n)(t) = \sum_n d_x(j,n)\psi_{j,n}(t),$$

where the coefficients $d_x(j,n)$ are defined through the inner product of the input signal with $\psi_{j,n}$,

$$d_x(j,n) = \langle x_n, \psi_{j,n} \rangle.$$

Based on the above, a resource signal can be represented without loss of information using the coarsest grain approximation signal and the underlying details. This is shown in the following relationship:

$$ResourceSignal, x_n = approx_J(t) + \sum_{j=0}^{J} detail_j(t)$$

The MRA analysis provides us with great flexibility. With this type of analysis, combinations of approximations and details can be studied together to better understand the dynamics of resource signals. As discussed earlier in the related work section of this paper, we performed an empirical study of the predictability of network bandwidth traces [16] using the Tsunami toolkit. In this

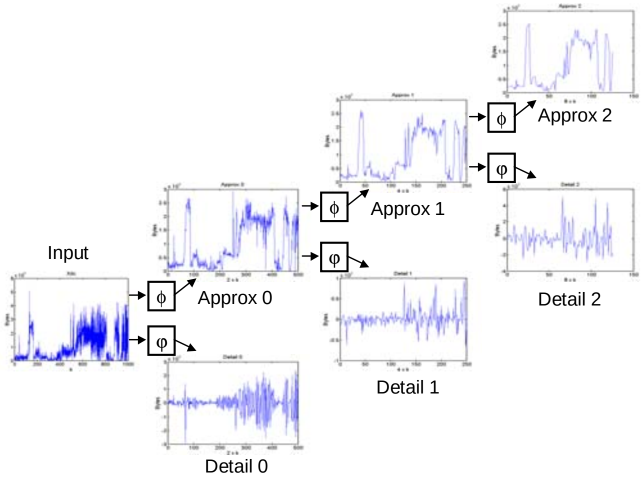


Figure 8: Multi-resolution analysis of three scales.

work, we first looked at the predictability of the detail signals and found that these signals are difficult to predict because they resemble that of *white noise*, a process lacking correlation structure. For an input signal that is mostly long-range dependent (LRD), a characteristic that binned network bandwidth traces tend to exhibit, it has been shown that the detail signals are mostly that of white noise, making it difficult to predict [8]. This work showed this for fractional brownian motion, an LRD process. However, this property is not true in general, and empirical studies can shed light on the predictability of the detail signals for an unspecified random process. From the failure of predictability on the detail signals, we looked into the predictability of the approximation signals and found that there exists some predictability. Many times there is a particular scale that proves to be the most predictable, a phenomenon that we coin the *predictability sweet spot*. The wavelet approximation signals are closely related to binning, a technique commonly employed to look at the multi-scale, multi-fractal properties of network bandwidth traces. The flexibility of an MRA analysis is extremely beneficial to analyzing the properties of resource signals, and may also prove to be important in online system building.

4 Goals and requirements

The goal of Tsunami is to facilitate two aspects of the construction of wavelet-based systems for use in distributed systems. The first is to provide a general wavelet system for analyzing resource signals using many different types of decompositions and basis functions. The second is to use the toolkit in building online components in distributed systems that lead to performance gains in prediction, dissemination and scalability.

4.1 Designing a wavelet system for research

In order to address the goals of the Tsunami toolkit, we must first understand the steps that a researcher would follow to use wavelets as a tool for analyzing resource signals and then proceeding to the building of online componenets. In what follows, we describe the steps a researcher would follow to accomplish the task of constructing an online system using wavelet based techniques.

1. Construct a sensor to measure the resource signal of interest. The measurements must be periodically sampled at a fine-grain rate such that all important characteristics of the resource is captured. If the rate is not fine-grain enough, a strange sort of aliasing may occur, in that the information captured will not be ground truth. The rate at which a resource is measured should be resource appropriate.

2. Create a trace file from the sensor output in order to facilitate offline analysis of the resource. This is important since one may go through many trials of analysis.

3. Decide upon the initial parameters of the offline wavelet analysis. These include the type of transform, the type of decomposition, the number of levels in the decomposition and the type of wavelet filter. This step may be repeated multiple times since there are degrees of freedom.

4. Qualitatively or quantitatively analyze the wavelet coefficients using other offline plotting tools and analysis tools such as Tsunami, Matlab, gnuplot or RPS to study the properties of the output representation.

5. Manipulate the wavelet coefficients using any technique that one sees fit. Examples are noise thresholding by zeroing out low energy coefficients, discarding various levels in the decomposition that are deemed unimportant, or performing various types of compression techniques.

6. Reconstruct the time-domain resource signal using a synthesis structure that matches the analysis structure in an appropriate way. Parameters between the analysis and synthesis structures should be matched in order to avoid unnecessary error.

7. Determine the success of the analysis by comparing the reconstructed time-domain signal with that of the original resource signal to obtain various performance metrics.

8. Construct an online system using the parameters of the study which had yielded the best results, and customize the system using home-grown solutions. Customized components are placed in front of the analysis section, between the analysis and synthesis sections or at the back of the synthesis section depending strongly on the application and the goals of the system.

A flowchart diagram of the process is shown pictorially in Figure 9.

To address 1, there are many sensors that exist today to measure resource signals in distributed systems. The RPS toolkit provides sensors which measure host load, network bandwidth, Windows performance data and /proc resource signals. The Remos [11] and the Network Weather Service (NWS) [26] systems provide sensors that measure available bandwidth between two endpoints. If the system is to be used under the Windows operating system, the Watchtower [10] system can be employed to measure hundreds of performance counters for generating multi-variate, periodically

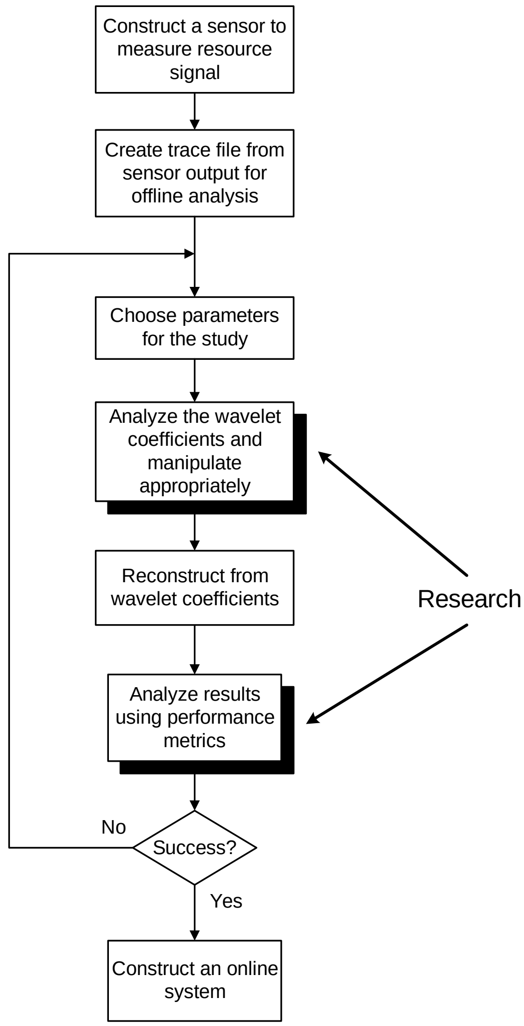


Figure 9: A flowchart of the steps of research using Tsunami.

Parameter	Description
Transform type	The transform type decides what kind of decomposition is performed. This can be a DWT, streaming wavelet transform, wavelet packets with entropy, uniform filter banks, or optimized low-delay filter banks. The transform can also be dynamically changing based on the signature of the incoming resource signal.
Number of levels	The number of levels in the decomposition is typically a function of the transform type and the frequency characteristics of the resource signal to be analyzed.
Wavelet basis function	Which basis function to use is typically decided upon empirically. However, high order basis functions increase the system delay, and therefore one would like to use the lowest order basis function while still achieving good results.

Figure 10: The degrees of freedom of a wavelet offline analysis.

sampled resource signals. Each of these utilities can write the measurements of the environment to file to be used for offline analysis, addressing item 2.

In item 3, there are many types of parameters that can be manipulated during wavelet offline analysis. Figure 10 lists what we believe to be the degrees of freedom in offline analysis. One must first decide what type of transform is to be used. Among the most common transforms are the DWT, wavelet packets, or any other type of frequency decomposition ranging from uniform to non-uniform and combinations between. The next parameter to be determined is the number of levels used in the decomposition. A resource signal which contains high energy in the low frequency bands will typically want to increase the number of levels in the decomposition, leading to narrow low-frequency bands. The basis function used, which represents the filter type and coefficients, are typically determined empirically. One can start with a Haar wavelet ($DAUB2$) and increase the filter order linearly to obtain the type of smoothing required for a given application. There are many degrees of freedom in a wavelet analysis, and researchers demand the flexibility to experiment with each of these parameters in their work.

In order to evaluate the parameters under test listed in 4, the output of the analysis (i.e. wavelet coefficients or reconstructed signals) are imported into other offline tools for performing statistical analysis or to look at the output of the decomposition in qualitative fashion. Tools that we have used for evaluating our parameter choice include Tsunami, Matlab, gnuplot and RPS. Gnuplot is a tool that can be used to perform simple graphing of data. However, a more powerful tool for graphing and also performing time-series analysis in a complete package is the Matlab software from Mathworks. Tools such as RPS can be used to analyze the predictability of wavelet coefficients to try and find performance gains in scheduling message transfers over network links or scheduling computation on a set of candidate hosts. Predictors can also be employed to attempt to reduce real-time system delay in the system.

As stated in items 5 and 6, the typical use of the wavelet transform is to manipulate the coefficients for a given application, and then perform the reconstruction. When building systems that use the wavelet transform, applications are typically placed between the transform block and the reconstruction block that solves some a-priori purpose. This application manipulates the coefficients in a way dictated by the application. The manipulated coefficients are then reconstructed, producing an estimate of the original resource signal.

As listed in 7, to determine the success of the simulation, a researcher will evaluate the performance of a given application or study by comparing the original input resource signal and the reconstructed resource signal in terms of some performance metric. If the metric deems the study unsuccessful, then step 3 can be repeated with different parameters guided by the value of the performance metrics under test. If the study is successful, then as step 8 suggests, a researcher may decide to build an online system to provide enhancements to their application by the inclusion of wavelet techniques.

Our goal is to provide a toolbox that accomodates the above steps. In order to create a tool for facilitating research using wavelets, we have come up with many design requirements of a wavelet-based research toolkit.

4.2 Design requirements of a wavelet-based system

To provide researchers with a powerful wavelet-based tool for furthering research in the understanding of resource signals and how they effect performance in distributed systems, we have compiled the following design requirements. The requirements are not ordered by importance, but rather from low-level to high-level abstractions.

Generality: The toolkit is designed so that many different communities can use Tsunami by simply creating a domain-based sample type based on their application. This sample abstraction should extend to blocks of samples so that samples may be aggregated and worked on in chunks. Communities that might benefit from this tool include graphics, robotics, network systems, interactive art and music.

Fine-granularity objects: The building blocks of the toolkit should be fine-grained so that many different types of structures can be constructed by appropriately connecting objects together. The tool must allow for arbitrary decompositions from uniform to non-uniform, and also tree-structures by the cascading of two-band filter-banks.

Extensibility in filtering: Many different types of wavelet basis functions (filter coefficients) should be supported, as well as different types of filters such as finite impulse response filters (FIR) and infinite impulse response filters (IIR). It should be easy to add new filters to the toolkit as the need arises.

Multiplicity in operation: Operations in the toolkit should run in sample by sample mode, or in block mode by the aggregation of samples into blocks. Mechanisms should be created which allow the user to aggregate samples into a block and call block operations, or clock samples into sample operation methods. In the transformations, state should be kept so that a user can switch between sample and block operation at execution time by calling the correct member function.

Streaming operation: We define streaming operation as the ability to clock samples or blocks of samples into the structure of the decomposition as soon as they are ready and have them clock out as they have been processed. Streaming operation treats the wavelet structure as a filter bank instead of just an algorithm as is the case with the DWT. When building online systems using wavelets, streaming operation is an invaluable technique when delay constraints are such that it is inappropriate to wait for a block of samples. This situation arises frequently in interactive systems and many times in distributed systems with real-time constraints.

Compatibility with other tools: The discrete wavelet transform (DWT) and the inverse discrete wavelet transform (IDWT) should be supported. In this type of transform, an input buffer is provided, length 2^M, yielding an output signal consisting of $M + 1$ levels, length 2^M. The parameter M is the number of two-band filter banks, a structure that we loosely term ***a stage***, used in the transformation. Since it is undefined to run the DWT/IDWT in streaming sample operation, this will only support block operations.

Flexibility in MRA: Interfaces should be constructed to obtain the approximation and detail signals. Any combination of the two types of signals should also be supported.

Adaptability: Time varying operation should be easily achieved through mechanisms of the design. The Tsunami design should support changing the number of bands in the decomposition, changing the decomposition type, and changing the type of filters used in the decomposition at run-time. This allows a researcher to adapt the structure of the analysis based on input signal dynamics.

Interoperability: The Tsunami toolkit must correctly interface to RPS, implying that the sensor outputs from RPS must be converted to samples that Tsunami can understand, and then back again. The interfaces must have the ability to be serialized over the network using RPS's mirror abstraction.

Jitter insensitivity: The tool should be able to recover from jitter and lost samples due to the infallability of the network. Recovery should attempt to do something reasonable such as interpolation (essentially averaging) or simply zeroing lost samples. If samples that arrive late have already been accounted for, the information in the late arriving sample can be used in the next jitter action event that takes place.

In the next section, we show how to obtain, compile and use Tsunami immediately as an analysis tool. Readers who wish to understand the software implementation before using the tool, can skip to Section 6 and later return to the next section for usage details.

5 Using Tsunami out of the box

We have built many command line utilities that allow researchers to immediately use Tsunami after it has been downloaded and compiled. Tsunami is made available through the release of RPS [2]. Once the release package has been downloaded and extracted from the tar file into the user defined root directory, all documentation related to the system build is located in the directory `doc`. The file `BUILD` provides an explanation on how to set the environment for an RPS build and how to compile the code for a given build environment. The Tsunami toolkit is highly dependent on the RPS build environment, and the steps outlined in the `BUILD` document must be followed initially. Once these steps are completed, `README.Wavelets` provides an explanation on how to build the Tsunami toolkit after all other build dependencies have been satisfied.

The Tsunami library is built from the *Wavelets* directory located as a sub-directory of the RPS root directory. The directory structure located in this directory follows from the build environment of RPS. The source code for the toolkit is located in the `include` and `src` directories. After successful compilation, the command line utilities will be in the directory

```
Wavelets/bin/$RPS_ARCH/$RPS_OS
```

accessed from the root directory of the release. The environment variables $RPS_ARCH and $RPS_OS describe the architecture of the machine and the operating system respectively. These environment variables are artifacts of releasing Tsunami with RPS and may be changed in future releases to its own independent project.

In this section, we provide examples of how to get started using Tsunami without learning most of the details needed to extend and build advanced applications with our toolkit. For a discussion on more advanced usage and extensions to the toolkit, we refer the reader to Section 7. In Figure 11 we list the statically structured, streaming utilities that we provide. Among these, we have utilities that compute the forward transformations providing wavelet coefficients and MRA coefficients as well as the reverse transformations that perform the reconstruction from wavelet coefficients. Each of these utilities can be run on individual samples or on a block of samples. Also, a mix

[2] To download the latest version of Tsunami, please visit the RPS website located at http://www.cs.northwestern.edu/∼RPS/

Utility Name	Description
Streaming Static Forward Transforms	
sample_static_sfwt	Forward static transform utility that provides approximation, detail and transform signals in sample operation
block_static_sfwt	Same as above in sample block operation
sample_static_mixed_sfwt	Forward static transform that provides a mix of approximation and detail signals based on a signal specification
block_static_mixed_sfwt	Same as above in sample block operation
Streaming Static Reverse Transforms	
sample_static_srwt	Reverse static transform utility that reconstructs the time-domain signal from wavelet coefficients
block_static_srwt	Same as above in sample block operation
sample_static_mixed_srwt	Reconstructions using a mix of approximation and detail signals based on a signal specification. May produce error between input signal and reconstruction based on mix of input signals
block_static_mixed_srwt	Same as above in sample block operation
Streaming Static System Tests	
sample_static_streaming_test	This utility performs a static forward transform and then reconstructs using a delay block and a reverse transform. An error signal is generated to show the system is working correctly. Error should be negligible
block_static_streaming_test	Same as above in sample block operation

Figure 11: Tsunami streaming static transform command line utilities

of approximations and details can be requested by using the mixed signal utilities. Test code is provided for determining whether the routines are working correctly after the build, and really provide no analytical benefits. The test routines compute the error between an input signal and the reconstructed signal. The error should be negligible if the toolbox has been installed correctly.

In Figure 12 we list the dynamic streaming utilities. The dynamic transforms are similar to the static transforms except that the structure of the decomposition and the coefficients used can be changed dynamically at runtime. The utilities listed here are extremely simple in that the changes happen at periodic points in time in terms of the number of samples. A more sophisticated application might detect epoch changes in the input signal and shape the structure or change the coefficients accordingly. This is an area that we have given some thought to, but we do not provide signal detection functionality in the current release of the toolkit.

In Figure 13 we list the discrete wavelet transform utilities. These utilities are a bit different from the others in that the transform is only to be run in block mode, and the block size is a function of the input signal length. Operations such as forward, reverse and mixed are supported in a similar manner to that of the static streaming and dynamic streaming transforms.

The arguments of each of these utilities are different based on the type of operation. We have recognized seven classes of command line arguments. These are *basic static streaming*, *mixed static streaming*, *basic dynamic streaming*, *mixed dynamic streaming*, *basic discrete*, *zerofill discrete* and *mixed discrete* command line arguments. In Figure 14, we compile a list of which utilities

Utility Name	Description
Streaming Forward Dynamic Transforms	
sample_dynamic_sfwt	Forward dynamic transform utility that provides approximation, detail and transform signals in sample operation. Structure and filter changes specified as a sample interval upon which to adapt
block_dynamic_sfwt	Same as above in sample block operation
sample_dynamic_mixed_sfwt	Forward dynamic transform that provides a mix of approximation and detail signals based on a signal specification. Changes occur based on sample change interval.
block_dynamic_mixed_sfwt	Same as above in sample block operation
Streaming Reverse Dynamic Transforms	
sample_dynamic_srwt	Reverse dynamic transform utility that reconstructs the time-domain signal from the forward transform. Upon a forward transform dynamic change, the reverse transform must change similarly. Change interval is passed as an argument
block_dynamic_srwt	Same as above in sample block operation
sample_dynamic_mixed_srwt	Reconstructions using a mix of approximation and detail signals based on a signal specification. Change interval is passed as an argument. May produce some error based on input signals
block_dynamic_mixed_srwt	Same as above in sample block operation
Streaming Dynamic System Tests	
sample_dynamic_streaming_test	The system test performs a forward dynamic transform followed by the appropriate delay component and the reverse dynamic transform. The structures of the forward and reverse change according to a sample interval passed as an argument.
block_dynamic_streaming_test	Same as above in sample block operation

Figure 12: Tsunami streaming dynamic transform command line utilities

belong to which class. The class designations are represented hierarchically in Figure 15.

The basic command line arguments for the streaming static transforms in the forward and reverse direction follow the form

```
./basic_static_streaming [input-file] [wavelet-type-init]
    [numstages-init] [transform-type] [flat] [output-file].
```

When the command is of a mixed signal type, a combination of approximation and detail signals, an additional signal specification file is required in the argument list. The format of the signal specificiation file is simple, and as an example we show the syntax for a user requesting five approximation signals and five detail signals. The signal specification file has the form:

Utility Name	Description
Discrete Transforms	
discrete_forward_transform	This utility performs the discrete wavelet transform on a block of samples length 2^M. It can provide the approximation, detail and transform signals from the operation
discrete_reverse_transform	This utility converts the encoded block of wavelet coefficients back into the time-domain signal.
discrete_reverse_zerofill_transform	Same as above, but zerofills levels according to a zero specification
discrete_forward_mixed	This utility performs the discrete wavelet transform and provides a mix of approximation and detail signals based on the signal specification
discrete_reverse_mixed	This utility converts back into the time-domain signal using a mix of approximation and detail signals. There may be some error involved in this operation
Discrete Transform System Tests	
discrete_transform_test	Performs a discrete wavelet transform followed by a reverse discrete wavelet transform. The input and output of this operation should be equivalent

Figure 13: Tsunami discrete wavelet transform command line utilities

Signal Specification File Format

```
# Signal type followed by whitespace followed by the number of
# levels in the specification and the level numbers.  This is
# used for mixed signal transforms.  Comments are designated
# by the ’#’ sign.  The form is:
# TYPE NUMLEVELS LEVELNUMBERS
APPROX 5 0 1 2 3 4         # set of approximation levels
DETAIL 5 0 1 2 3 4         # set of detail levels
```

If the requested signal levels do not make sense based on the total number of stages input to the command line utility, then the levels that can be satisfied are returned to the user. In these types of operations, the transform type has been excluded since an MRA analysis is assumed by the addition of the signal specification. The command line arguments for a mixed transform has the form

```
./mixed_static_streaming [input-file] [wavelet-type-init]
    [numstages-init] [specification-file] [flat] [output-file].
```

The streaming dynamic transforms have additional arguments over the static transforms in order to specify the frequency with which the structure and filter coefficients should change. The command line arguments for the dynamic utilities follow the form

```
./basic_dynamic_streaming [input-file] [wavelet-type-init]
    [numstages-init] [transform-type] [wavelet-type-new]
    [numstages-new] [change-interval] [flat] [output-file].
```

Argument class and list of utilities
Basic static streaming
sample_static_sfwt
block_static_sfwt
sample_static_srwt
block_static_srwt
sample_static_streaming_test
block_static_streaming_test
Mixed static streaming
sample_static_mixed_sfwt
block_static_mixed_sfwt
sample_static_mixed_srwt
block_static_mixed_srwt
Basic dynamic streaming
sample_dynamic_sfwt
block_dynamic_sfwt
sample_dynamic_srwt
block_dynamic_srwt
sample_dynamic_streaming_test
block_dynamic_streaming_test
Mixed dynamic streaming
sample_dynamic_mixed_sfwt
block_dynamic_mixed_sfwt
sample_dynamic_mixed_srwt
block_dynamic_mixed_srwt
Basic discrete
discrete_forward_transform
discrete_reverse_transform
discrete_transform_test
Zerofill discrete
discrete_reverse_zerofill_transform
Mixed discrete
discrete_forward_mixed
discrete_reverse_mixed

Figure 14: Argument classes and corresponding Tsunami utilities

As above, if the transform is of the mixed type and dynamic, then the command line arguments for these type of utilites follow the form

```
./mixed_dynamic_streaming [input-file] [wavelet-type-init]
    [numstages-init] [specification-file] [wavelet-type-new]
    [numstages-new] [change-interval] [flat] [output-file].
```

The dynamic mixed transforms could have an additional signal specification for each structure interval, but at this time we have provided just the base dynamic operation. It is a simple extension that requires very little additional code to add this functionality.

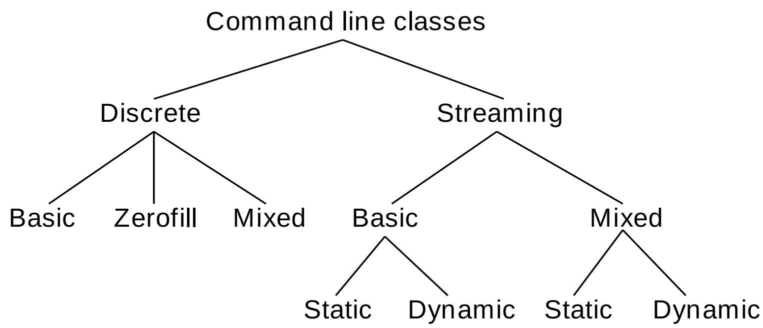


Figure 15: Command line utility class hierarchy.

The discrete wavelet transforms actually require less arguments than the streaming transforms. This is due to the fact that the number of stages are inferred from the length of the input file. The command line arguments for transforms of this type follow the form

```
./basic_discrete [input-file] [wavelet-type-init]
    [transform-type] [flat] [output-file].
```

A reverse transformation with some of the levels zero-filled is supported by the toolkit. This type of reverse transform requires a zero-fill specification file that designates which levels are to be zero filled before performing the reverse transformation. The file format is similar to that of the signal specification file shown earlier. It contains a zero-fill designator, the number of levels to zero and the level numbers. An example of zeroing out levels 0, 1, 4 and 5 are shown as follows:

Zero-fill Specification File Format

```
# This is the zero specification file for performing
# zero fill reverse transforms.  The form is:
# ZERO_DESIG NUMLEVELS LEVELNUMBERS
Z 4 0 1 4 5     # Zero out levels 0, 1, 4 and 5
```

When using the discrete reverse transform with the zero-filling of specific designated levels, the command line arguments follow the form

```
./zerofill_discrete [input-file] [wavelet-type-init]
    [zerospec-file] [transform-type] [transform-type] [flat]
    [output-file].
```

The discrete transform can also be run in mixed mode, which in that case it would be run with the addition signal specification argument. The command line arguments for mixed signal commands of this type follow the form

```
./mixed_discrete [input-file] [wavelet-type-init]
    [specification-file] [flat] [output-file].
```

In Figure 16, a description of the arguments that we support for static and dynamic streaming transforms and the discrete block transforms are shown. The figure lists the argument classes and

		Streaming				*Discrete*		
		Static		*Dynamic*				
Argument	Description	Basic	Mixed	Basic	Mixed	Basic	Mixed	Zerofill
input-file	Formatted input file of samples	yes	yes	yes	yes	yes	yes	yes
wavelet-type-init	The wavelet basis function to use	yes	yes	yes	yes	yes	yes	yes
numstages-init	The number of levels in the decomposition	yes	yes	yes	yes	no	no	no
transform-type	Approximation only (APPROX), Detail only (DETAIL), or Transform (TRANSFORM)	yes	yes	yes	yes	yes	yes	yes
specification-file	Used for mixed signals to specify which approximations and details to output	no	yes	no	yes	no	yes	no
zerospec-file	Used for zero-filling levels in discrete reverse transforms	no	no	no	no	no	no	yes
output-file	Formatted output file of wavelet coefficients or reconstructed samples	yes	yes	yes	yes	yes	yes	yes
flat	Designates whether the output should be human readable or not	yes	yes	yes	yes	yes	yes	yes
wavelet-type-new	The wavelet basis function to dynamically switch into place	no	no	yes	yes	no	no	no
numstages-new	The new number of stages to dynamically switch to	no	no	yes	yes	no	no	no
change-interval	The amount of time in samples before changing to the new wavelet types and number of stages	no	no	yes	yes	no	no	no

Figure 16: Tsunami command line arguments

which arguments belong to each class. In the class list, *yes* implies that the argument is required for that class of input arguments while *no* implies the opposite. From the two figures, it is important to note that our system test codes and reverse transforms belong to the basic argument classes but only take TRANSFORM as the transform-type. The test utility is to observe the perfect reconstruction property of the transform, and for the reverse transform utilities it doesn't make much sense to reconstruct using only detail or approximation signals.

To provide the reader with a flavor of a sample utility, in the following we show the source code of one of the basic system tests, the *sample_static_streaming_test* module. We choose to show this code because it demonstrates how to use the streaming forward transforms, the delay block required for perfect reconstruction in a streaming transform and the streaming reverse transforms. This utility as well as all other utilities are written in the C++ programming language.

Usage for utility

```
void usage()
{
  char *tb=GetTsunamiBanner();
  char *b=GetRPSBanner();

  cerr << " sample_static_streaming_test [input-file] [wavelet-type-init]\n";
  cerr << "  [numstages-init] [transform-type] [output-file]\n\n";
  cerr << "-------------------------------------------------------------\n";
  cerr << "\n";
  cerr << "[input-file]        = The name of the file containing time-\n";
  cerr << "                      domain samples.  Can also be stdin.\n";
  cerr << "\n";
  cerr << "[wavelet-type-init] = The type of wavelet.  The choices are\n";
  cerr << "                      {DAUB2 (Haar), DAUB4, DAUB6, DAUB8,\n";
  cerr << "                      DAUB10, DAUB12, DAUB14, DAUB16, DAUB18,\n";
  cerr << "                      DAUB20}.  The 'DAUB' stands for\n";
  cerr << "                      Daubechies wavelet types and the order\n";
  cerr << "                      is the number of coefficients.\n";
  cerr << "\n";
  cerr << "[numstages-init]    = The number of stages to use in the\n";
  cerr << "                      decomposition.  The number of levels is\n";
  cerr << "                      equal to the number of stages + 1.\n";
  cerr << "\n";
  cerr << "[transform-type]    = The transform type may only be of type\n";
  cerr << "                      TRANSFORM for this test.\n";
  cerr << "\n";
  cerr << "[output-file]       = Which file to write the output.  This\n";
  cerr << "                      may also be stdout or stderr.\n\n";
  cerr << tb << endl;
  cerr << b << endl;
  delete [] tb;
  delete [] b;
}
```

Parse input arguments and define types

```
int main(int argc, char *argv[])
{
  if (argc!=6) {
    usage();
    exit(-1);
  }

  istream *is = &cin;
  ifstream infile;
  if (!strcasecmp(argv[1],"stdin")) {
  } else {
    infile.open(argv[1]);
    if (!infile) {
      cerr << "sample_static_streaming_test: Cannot open input file "
           << argv[1] << ".\n";
      exit(-1);
    }
    is = &infile;
  }

  WaveletType wt = GetWaveletType(argv[2], argv[0]);

  int numstages = atoi(argv[3]);
  if (numstages <= 0) {
    cerr << "sample_static_streaming_test: Number of stages must be "
         << "positive.\n";
    exit(-1);
  }

  if (toupper(argv[4][0])!='T') {
    cerr << "sample_static_streaming_test: For streaming tests, "
         << "only TRANSFORM type allowed.\n";
    exit(-1);
  }

  ostream *outstr = &cout;
  ofstream outfile;
  if (!strcasecmp(argv[5],"stdout")) {
  } else if (!strcasecmp(argv[5],"stderr")) {
    outstr = &cerr;
  } else {
    outfile.open(argv[5]);
    if (!outfile) {
      cerr << "sample_static_streaming_test: Cannot open output file "
           << argv[5] << ".\n";
      exit(-1);
    }
    outstr = &outfile;
  }
```

Read input data from filestream

```
// Read the data from file into an input vector
vector<wisd> samples;
FlatParser fp;
fp.ParseTimeDomain(samples, *is);
infile.close();
```

Instantiate classes and setup result containers

```
// Instantiate a static forward wavelet transform
StaticForwardWaveletTransform<double, wosd, wisd> sfwt(numstages,wt,2,2,0);

// Parameterize and instantiate the delay block
unsigned wtcoefnum = numberOfCoefs[wt];
int *delay = new int[numstages+1];
CalculateWaveletDelayBlock(wtcoefnum, numstages+1, delay);
DelayBlock<wosd> dlyblk(numstages+1, 0, delay);

// Instantiate a static forward wavelet transform
StaticReverseWaveletTransform<double, wisd, wosd> srwt(numstages,wt,2,2,0);

// Create result buffers
vector<wosd> outsamples;
vector<wosd> delaysamples;
vector<wisd> finaloutput;
vector<wisd> outsamp;
```

Perform operations

```
for (unsigned i=0; i<samples.size(); i++) {
  sfwt.StreamingTransformSampleOperation(outsamples, samples[i]);
  dlyblk.StreamingSampleOperation(delaysamples, outsamples);
  if (srwt.StreamingTransformSampleOperation(outsamp, delaysamples)) {
    for (unsigned j=0; j<outsamp.size(); j++) {
      finaloutput.push_back(outsamp[j]);
    }
  }

  outsamp.clear();
  outsamples.clear();
  delaysamples.clear();
}
```

```
                          Produce the output
  for (unsigned i=0; i<MIN(finaloutput.size(), samples.size()); i++) {
    *outstr << i << "\t" << samples[i].GetSampleValue() << "\t"
                  << finaloutput[i].GetSampleValue() << endl;
  }
  *outstr << endl;

  // Calculate the error between input and output
  double error=0;
  unsigned sampledelay =
    CalculateStreamingRealTimeDelay(wtcoefnum,numstages) - 1;
  unsigned i=0, j;
  for (j=sampledelay; j<MIN(finaloutput.size(), samples.size()); i++, j++) {
    error += samples[i].GetSampleValue() - finaloutput[j].GetSampleValue();
  }

  *outstr << "Mean error: " << error/(double)i << endl;
```

```
                                Clean up
  // Destruct allocated memory
  if (delay != 0) {
    delete[] delay;
    delay=0;
  }

  return 0;
}
```

Most utilities that we have created, have a similar structure to the code example listed above. Most of the difference occurs because of dynamic, mixed signal or discrete operation. Each other streaming utility that is provided in the Tsunami toolkit is a subset of what we have shown above. The discrete transforms are block mode transforms, and are different than what is shown.

The structure of the code will become more clear after we have discussed the software implementation and design in the next section. The reader may want to come back to the above code example after reading the next section.

6 Design and Implementation

In this section, we discuss the software design and implementation of the Tsunami toolkit. In order to lead the discussion, Booch diagrams [3] are shown with the important attributes listed for each class. Hierarchical representations are provided when needed.

The overall software structure of the Tsunami toolkit is shown in Figure 17. The figure shows the objects that are created in order to create wavelet type transforms and arbitrary decompositions from these blocks. What is not shown in the figure is the sample and sampleblock representations used for shipping around periodic samples, and the discrete type of transforms. The discrete transforms can be looked at as tree structured, but in our design, the discrete transforms are all inclusive. It simply executes the algorithm for performing the DWT and the IDWT. We would like to direct the reader to the similarity between this figure, and that shown earlier in Figure 6(a).

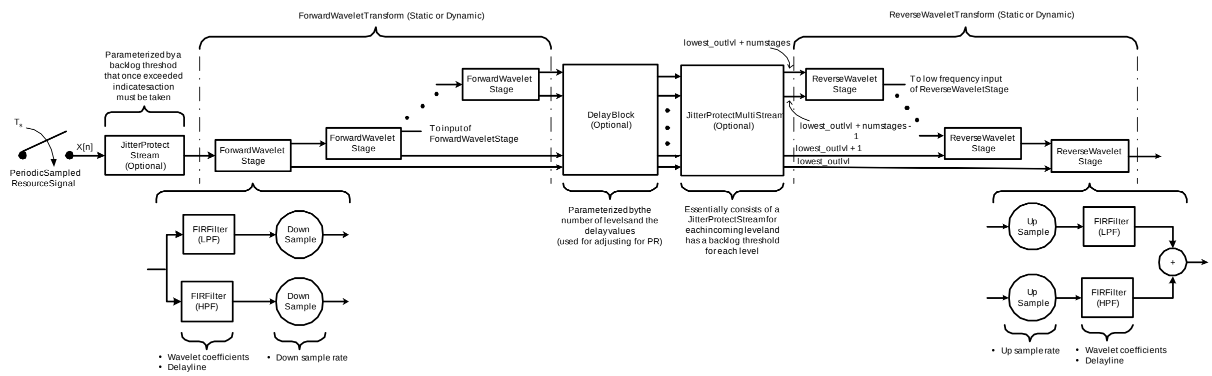


Figure 17: System software design.

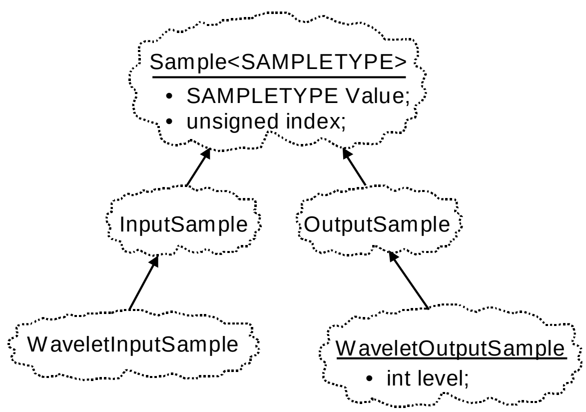


Figure 18: The sample class hierarchy.

A listing of the class interfaces with a description of the member functions that each provide is in Appendix A.

6.1 Generic design starting with samples

In order to allow the design to be general for many types of users and communities, the toolkit is built using the C++ generic class mechanisms and inheritance. In Figure 18 we show the sample class hierarchy. At the lowest level of the toolkit there is the *Sample* base class that is generically typed by a sample type. This class provides many operators to manipulate samples, such as adding two samples together, setting the values of the sample, and getting the value of a sample. The data attributes of this class include the sample value, and the sample index. The sample index assumes that each sample is equally spaced apart. Typically output samples are resampled to a lower sample rate than that of the input sample rate in order to reduce redundancy in the representation. This was discussed in more detail in Section 3.

Inherited from the *Sample* base class, are two classes called *InputSample* and *OutputSample*. In most situations, when performing wavelet transform operations the input samples and output samples are annotated differently. For instance, the output samples of a wavelet transform have some notion of level of the decomposition unless the level is encoded in the block ordering of the

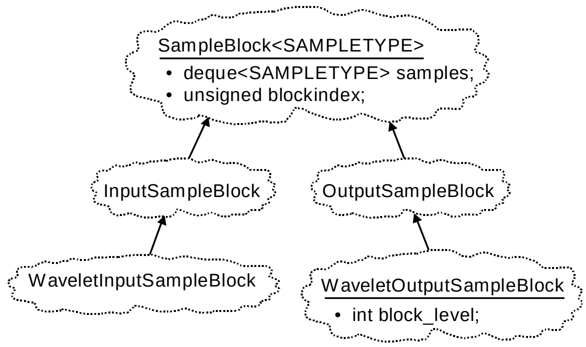


Figure 19: The sample block class hierarchy.

samples. The input and output sample classes serve as the split between the two types of samples.

One level down in the hierarchy, are the classes *WaveletInputSample* and *WaveletOutputSample*. The *WaveletInputSample* class is subclassed from *InputSample*$<$*SAMPLETYPE*$>$. In our applications, the *SAMPLETYPE* is of type *double*. The *WaveletOutputSample* class is a subclass of *OutputSample*$<$*SAMPLETYPE*$>$ where the *SAMPLETYPE* is also of type *double*. The output samples have the additional *level* annotation. The level of the output sample is set once wavelet transformations are perfomred, and is meaningless until then. The level of the output sample is assigned starting with the highest frequency band designated by *lowest_outlvl*, a parameter of the transform operation. The level number increases as the sample represents lower frequency information. This is explicitly shown in Figure 17.

If a user would like to create their own sample type, this is done by subclassing from the *InputSample* and *OutputSample* classes.

6.2 Aggregating samples into blocks

Since one of the requirements of the toolkit is to perform block operations on aggregated samples, we have created the *SampleBlock* data type and its subclasses. This is shown in Figure 19.

The *SampleBlock* class serves as the base class of the block datatypes. In the typical sense but not restricted to, the *SampleBlock* class is typed by our *Sample* class discussed previously. It uses the C++ Standard Template Library (STL) *deque* container class for aggregating blocks of samples. The reason for using the *deque* data structure, is due to the fact that some of the transformations and algorithms implemented in the toolkit contain data access patterns that add samples to the beginning and to the end. The other data member of the *SampleBlock* class is the block index. Even though samples as represented by the *Sample* class contain a data member for indexing, it is much more efficient when working with sample blocks to have a block index instead of having to peer in at the samples directly. There is an underlying assumption to using sample blocks, and that is that each of the samples contained within the block are in order and there are no missing samples within the block.

The *SampleBlock* class provides many interfaces for working with blocks of samples. These include member functions for obtaining specific samples, obtaining a subset of the samples, pushing and popping samples into and out of the block, and adding two blocks together. These abstractions make it easy to work with blocks in the context of filtering, re-sampling and transforming

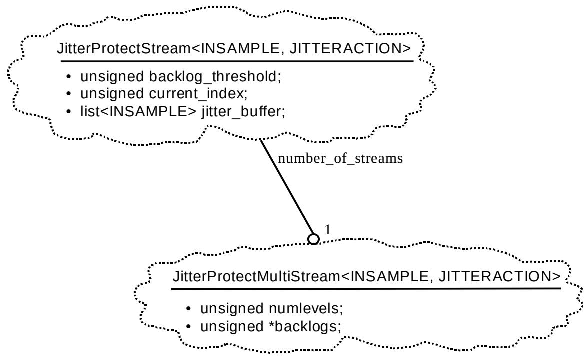


Figure 20: The jitter protection classes.

blocks into the wavelet domain.

The structure of the *SampleBlock* hierarchy follows similarly to that of the *Sample* class hierarchy. There are two subclasses of *SampleBlock*. These are the *InputSampleBlock* and the *OutputSampleBlock*. Subclassed from these are the *WaveletInputSampleBlock* and the *WaveletOutputSampleBlock*. The *WaveletInputSampleBlock* is an *InputSampleBlock*, but parameterized by the sample class *WaveletInputSample*$<$*double*$>$. The *WaveletOutputSampleBlock* is an *OutputSampleBlock*, but it contains the extra attribute to designate the *level* of the decomposition the block of samples represent.

Like the sample classes, if one wants to make a different sampleblock type for a different purpose than our specific purpose, they may simply subclass off of *InputSampleBlock* and *OutputSampleBlock* to address their particular needs. We believe that the sample and sample block structure that we have created is generic to the extent that any type of sample for any type of purpose can be created within the framework that we provide.

An example of how to work with these classes as related to our community of analyzing resource signal samples in distributed computing is shown as follows

```
// Create a type definition for input and output samples
typedef WaveletInputSample<double> wisd;
typedef WaveletOutputSample<double> wosd;

//Create some input and output blocks of samples
WaveletInputSampleBlock<wisd> inputblock;
WaveletOutputSampleBlock<wosd> outputblock;
```

Once the *wisd* and *wosd* types have been created, it is very easy to type other operations that we will discuss in the following class descriptions.

6.3 Jitter protection

Because Tsunami is a system built for use in distributed computing and because, in this domain, samples are typically sent over an unreliable network, the system must appropriately deal with loss, corruption and samples arriving late. In order to deal with these problems, Tsunami provides jitter components for handling all of these cases.

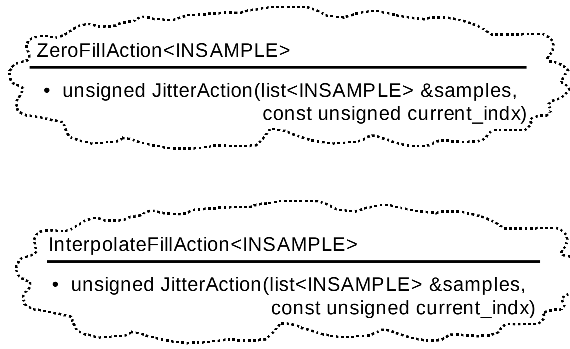


Figure 21: The jitter action classes.

If Tsunami is used with TCP, corruption and loss is dealt with appropriately in the TCP protocol stack. However, the sample arrival times are not guaranteed in TCP and therefore the system must deal with the samples that arrive late. The varying arrival times of samples is known as *jitter*. In addition, if used with UDP, there is no guarantee that samples will arrive at all, and the system will need to deal with lost samples also. To deal with these types of reliability problems, the toolkit provides a set of classes to handle jitter and loss in the network in the appropriate manner.

The jitter protection classes that we provide are shown in Figure 20. The class *JitterProtectStream* can protect streams of samples or streams of blocks being sent over the network for processing. This is typically the case when the machine running the sensor simply collects measurements and sends them off to another machine for computing the transformations. This class contains a jitter buffer for reordering the samples, an index of the sample that is next expected from the network, and a backlog threshold that determines when to take the appropriate action for fixing the jitter problem. The class *JitterProtectMultiStream* protects multiple streams of samples by using level information and sample or block indices. This class simply uses the class *JitterProtectStream* for each of its multiple streams. The *JitterProtectMultiStream* has an array of backlog thresholds for each stream that it is protecting. When a particular threshold has been exceeded, missing samples must be filled appropriately before processing can continue. Because there is a notion of perfect reconstruction in wavelet analysis, jitter recovery is important for reducing the error between the input resource signal and the reconstructed signal.

When either of these components sense jitter and or an extreme loss of samples, the jitter action routines are called in order to keep the system moving forward. The jitter action classes are shown in Figure 21. The first action class, *ZeroFillAction*, zero fills missing samples so that the system can progress. The second action class, *InterpolateFillAction*, will fill in samples according to an average over the samples received thus far. Users can extend the toolkit by adding other jitter action classes by parameterizing *JitterProtectStream* and *JitterProtectMultiStream* by a jitter action class. Each new jitter action class that is built, will create a *JitterAction* member function that takes as arguments the current index and an STL *list* data structure for the newly created output samples. As an example, to protect *WaveletInputSamples* from jitter as they are shipped to another machine for transformation processing, and using the *ZeroFillAction* class for sample recovery, one would instantiate a stream by the following:

```
// Instantiate a jitter protection class on a
//  WaveletInputSample stream using ZeroFillAction
```

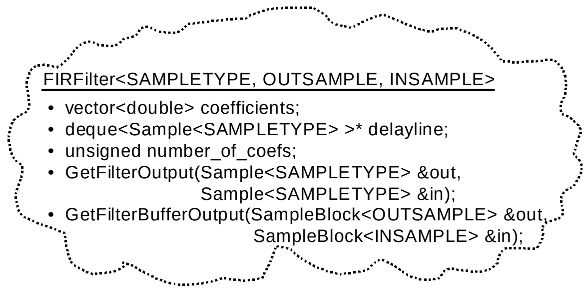


Figure 22: The FIR filter class.

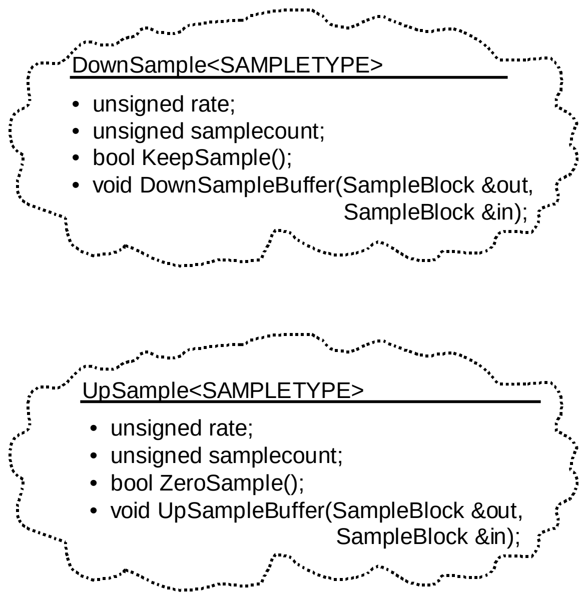


Figure 23: The up and down sample classes.

```
typedef WaveletInputSample<double> wisd;
JitterProtectStream<wisd, ZeroFillAction<wisd> > jps;
```

Although jitter protection is a much bigger requirement for Tsunami than for RPS, this set of classes can be used for jitter protection for standard RPS communication as well.

6.4 Fine-grain building blocks

To address the goal of fine-grain building blocks, we chose to split up each of the processing components into fine-grained modules so that many different types of structures can be constructed. The objects that we discuss next are shown in the simple two-band filter bank example in Figure 5(b). In order to create the structure shown in the figure, we need to implement filters of various characteristics including low-pass and band-pass responses, decimators and expanders. In this report, we use decimators and downsamplers, and expanders and upsamplers interchangeably.

In Figure 22, we show an FIR filter implementation parameterized by the sample types contained in the delay line of the filter, and the input and output sample types of the operation. Attributes of the FIR filter include the coefficients that characterize the filter, the delay line of the filter, and the number of coefficients used in the particular filter. Since the system is based on wavelets and we have only implemented the Daubechies designed filters [4], we currently only have support for FIR filters. The filter class can be used in sample operation or block operation, following the overall goal of multiplicity in operation.

In Figure 23, we show the up and down sampling classes. The decimator section of the structure is implemented in the *DownSample* class and is parameterized by the type of sample that will be input and later output. The assumption with the *DownSample* class, is that whatever sample type is input will also be output. The down sample operation can be run at any down sample rate which is typically a function of the type of decomposition. In the two-band structure shown in the figure, we down sample by two. The operation can be run on samples by using the member function *KeepSample()* or in block mode with the function *DownSampleBuffer()*.

The expander section is implemented in the *UpSample* class and is parameterized by the type of sample, similar to the *DownSample* class. It contains the same assumptions related to the parameterization of input and output samples as the *DownSample* class. The up sample class can be run at any rate and is also a function of the type of structure. In the two-band figure decomposition, the up sample rate is two. Similarly to the *DownSample* class, the *UpSample* class can be run on samples or on blocks of samples.

6.5 Support for many filters

In order to support the goal of extensibility in filtering, we would like to support many different types of filters. At this time, however, we have only implemented the FIR filter type. Other types of filters that we may like to have in the future include infinite impulse response (IIR) filters and paraunitary block filters that provide for greater computational efficiency. Due to the generic structure of our filter design, and how these objects fit into the overall structure of a transformation, any type of filtering operation can be performed on input or output samples.

We currently have support for the wavelet coefficients designed by Daubechies, from DAUB2 (the Haar wavelet) to DAUB20. Since the Daubechies coefficients are constrained to even order, we currently provide ten different types of wavelet filters. As the Daubechies wavelet filters increase in order, the decomposition tends to smooth. However, there is a tradeoff between smoothness and system delay, which we will discuss further in Section 8.

We have also implemented two examples of low-delay filter bank filters with different delay signatures copied directly from the work done by K. Nayebi et al [14]. We primarily took examples of low-delay filters from this paper to validate its claims. Our future plans is to implement the low-delay filter bank algorithm in the Tsunami toolkit in order to provide the same functionality with low-delay. Low-delay operation is extremely important in general for many applications, but especially for interactive applications. The algorithm designs uniform and non-uniform low-delay filter banks based on the number of bands required in the decomposition, the number of coefficients in each filter, the frequency response of the analysis filters, and the overall system delay. These parameters are then input into an optimization procedure which, after convergence, provides the filter coefficients for the analysis and synthesis filters. There are other types of extensions that might also prove beneficial, such as the modulated filter bank techniques found by G. Schuller and T. Karp [20].

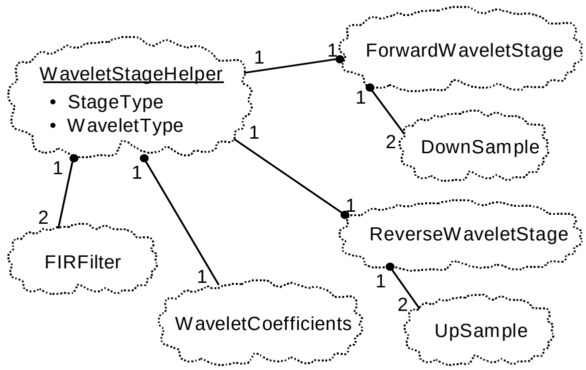


Figure 24: The stage relationships.

Our hope is that the interfaces that we have provided and the general organization of the toolkit is such that other users of the system find it easy to implement their own types of filters to fit their needs. This is done by simply adding coefficients to the source file *coefficients.cpp*, and instantiating the appropriate filter type with the newly added coefficients. More elaborate details are given on how to add filters and filter types in Section 7.

6.6 Stages

From the implementation of filters, decimators and expanders, many different types of structures can be built. We have built forward stages and reverse stages for use in tree-structured decompositions from these fine-grained objects. The Booch diagram of a stage is shown in Figure 24. Each stage class, *ForwardWaveletStage* and the *ReverseWaveletStage* each contain one helper object *WaveletStageHelper*. This helper class contains the commonality of both the forward and reverse stages, namely FIR filters and the coefficients of these filters. The class *ForwardWaveletStage* customizes the wavelet stage helper to provide the analysis filters and also contains the two down samplers. The stage abstraction also provides a notion of the output level numbers that it is responsible for in order to avoid ambiguity when we chain stages together to create various decompositions. The class *ReverseWaveletStage* contains one wavelet stage helper customized with the synthesis filters and also two up samplers. This stage type does not need any notion of level at the output, because it is typically used to convert back to a one-dimensional time-domain resource signal.

From the stages, we then create tree-structured decompositions as shown in Figure 6(a), or other more balanced tree structures for other decompositions of the resource signal. Currently, Tsunami supports classes for creating the tree-structure types shown by the chaining of two-band stages together. However, the design is general enough for any other type of structure that one might want to create.

6.7 Transforms

All decompositions except for the DWT and IDWT operations have been created to handle running transforms in sample or in block transform mode, thus satisfying the goal of multiplicity in

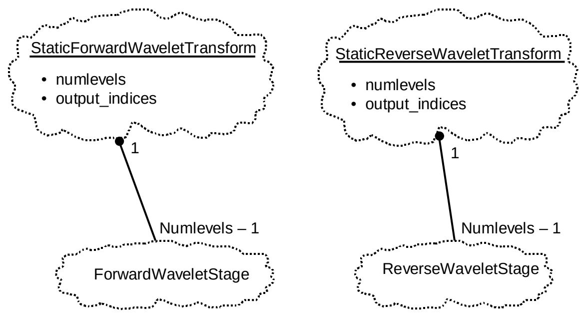


Figure 25: The static transform classes.

operation. In the sections that follow, we discuss the various types of transforms provided in the toolkit.

6.7.1 Static transforms

Figure 25 shows how static transforms in forward and reverse direction are constructed. Forward directed transforms take as input a one-dimensional resource signal and produce as output the wavelet coefficients. Reverse directed transforms take as input the wavelet coefficients and produce as output the reconstructed one-dimensional signal. Each transform contains $numlevels - 1$ stages, producing a total of $numlevels$ of approximation and detail signals. The number of levels in the decomposition is highly dependent on the dynamics of the input resource signal. The *StaticForwardWaveletTransform* class, is instantiated with the number of stages, the wavelet basis function type, the down sample rates and the lowest output level in the decomposition. Once the structure has been instantiated, the transform can be run in sample or block operation mode by calling the correct member function.

The *StaticReverseWaveletTransform*, is instantiated with the number of stages, the wavelet basis type, the up sample rates and the lowest input level of the samples or blocks streaming into the structure. Once instantiated, the reverse transforms can be run in sample or block operation, and also contains functions for zeroing out levels that are deemed unimportant in the reconstruction.

A more detailed description of the member functions for the static transform classes are listed in Appendix A.

6.7.2 Dynamic transforms

As shown in Figure 26, the dynamic transforms are subclassed from the static transforms. These transforms are constructed to specifically address the requirement for time-varying operation, and have additional member functions over the static transforms for this. Each of the dynamic transforms, the *DynamicForwardWaveletTransform* and the *DynamicReverseWaveletTransform*, contain member functions for adding and removing stages and changing the wavelet basis functions at various stages. These changes in structure and operation can be made at run-time by calling the appropriate operations. At this time, when a stage is added or removed or the wavelet basis function is changed, there is an associated transition error seen between the output of the reverse wavelet transform and the input resource signal. In order to combat this transitional error, the work

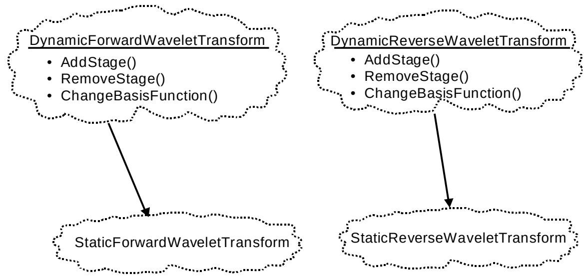


Figure 26: The dynamic transform classes.

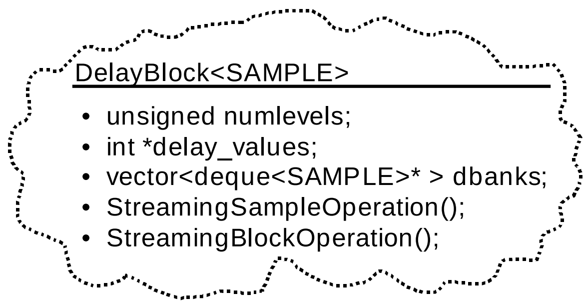


Figure 27: The delay block class.

implemented by I. Sodagar, K. Nayebi and T. P. Barnwell on time-varying filter banks and wavelets should be implemented [23].

6.7.3 The delay block

Figure 17 shows where the delay block is placed in the software diagram, and Figure 27 describes the implementation of the class. The delay block is required to phase align the various levels of the streaming transform in order to achieve the perfect reconstruction property. This block is not required for the discrete transforms. Without this block, the low level, high frequency bands, are filtered through the structure faster then the higher level, low frequency bands, and are not properly phase aligned. This block does the re-aligning required to achieve perfect reconstruction.

6.7.4 Discrete wavelet transforms

Figure 28 shows the discrete transforms that are currently supported in the toolkit. Each of these transforms have data members for the wavelet type (i.e. Haar, D4, etc.) and the coefficients of this type for implementing the discrete algorithms. These transforms are different from the other two that we have discussed thus far, in that there is an implicit assumption relating the input to the output. The assumption of the *ForwardDiscreteWaveletTransform* is that it takes as input a *SampleBlock* of length 2^M, and produces as output a block of samples that represent $M + 1$ levels of wavelet coefficients. The output block that is generated is specially encoded in the class *DiscreteWaveletOutputSampleBlock*, a subclass of *OutputSampleBlock*. The encoding is shown in Figure 29. The *ReverseDiscreteWaveletTransform* takes as input the *DiscreteWaveletOutputSampleBlock* and reconstructs back into the type *SampleBlock* representing the reconstructed

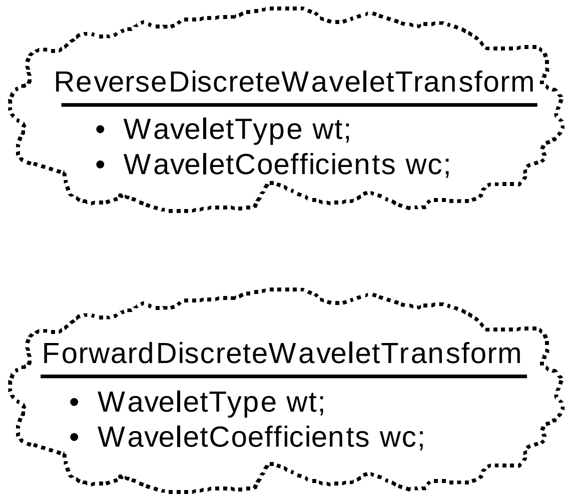


Figure 28: The discrete transform classes.

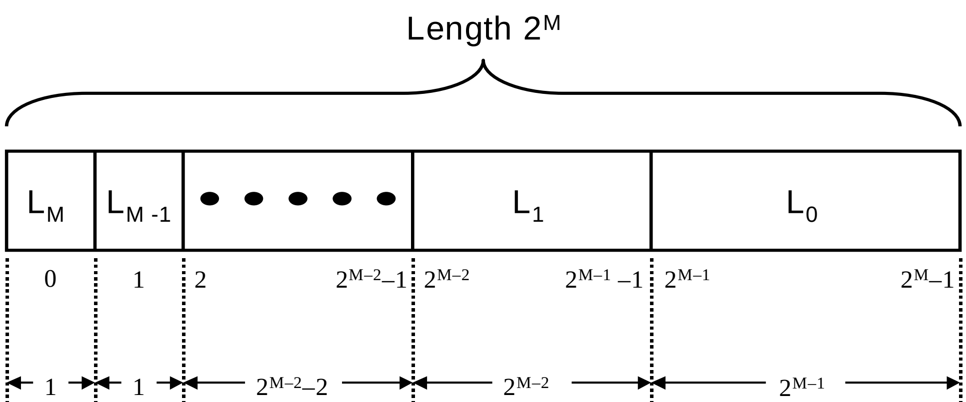


Figure 29: The encoding of the DiscreteWaveletOutputSampleBlock. Shown are the indices of the block and the lengths of each segment. The levels are designated by L_i, where i is the level number.

time-domain signal.

Following from the other transforms discussed earlier, the DWT can produce all of the approximation and detail signals, as well as the transform representation of one approximation and a set of details, and the mixed signal representations. For more details on the specific functions in the discrete classes, refer to appendix A.

From the requirements and goals of the Tsunami toolkit, we have created a first version implementation. There is still much work to be done in order to completely fulfill these goals, but as a first release, most requirements have been satisfied. The implementation has a flavor of the complete tool that we would like to eventually have, and will only become more closely matched to the goals of the project over time.

7 Advanced usage and extensions

In this section, we will describe how to use Tsunami built in interfaces to construct decompositions provided in the toolkit. From the following code examples, it should become clear how to piece together more sophisticated decompositions with the provided building blocks. We start by describing the code contained within the *StaticForwardWaveletTransform* for decomposing the signal into a non-uniform decomposition. From here we show the code for instantiating the reconstruction structure using the *StaticReverseWaveletTransform* code. This provides the reader with a concrete example of how to construct decompositions using Tsunami building blocks.

In the second part of this section, we discuss extensions to the Tsunami toolkit. Wavelet packets (a more general wavelet structure), time-varying operation and filter extensions are discussed.

7.1 Using Tsunami building blocks for advanced decompositions

In this section, we discuss how to instantiate a static wavelet transform by using the code contained in the *StaticForwardWaveletTransform* class. From the stage class, many other types of decompositions can be created. We will also show a reconstruction using reverse stages from the code contained in the *StaticReverseWaveletTransform* class.

The constructor code that follows takes as input the number of stages in the decomposition, the wavelet filter type, the downsample rates on the low-pass and high-pass branches and the lowest output level. The code performs checks as to the sanity of the number of stages, and initializes protected data members for indexing and output level annotation. Next, the first stage in the decomposition converts the *INSAMPLE* type to the *OUTSAMPLE* type. The remaining stages work with *OUTSAMPLE* types only. The chaining of stages in this code is represented as a vector of stages.

Non-uniform decomposition using forward stages

```
template <typename SAMPLETYPE, class OUTSAMPLE, class INSAMPLE>
StaticForwardWaveletTransform<SAMPLETYPE, OUTSAMPLE, INSAMPLE>::
StaticForwardWaveletTransform(const unsigned numstages,
                              const WaveletType wavetype,
                              const unsigned rate_l,
                              const unsigned rate_h,
                              const int lowest_outlvl)
{
  unsigned i;

  // Argument checks and data initializations
  if ( (numstages == 0) || (numstages > MAX_STAGES) ) {
    this->numstages = 1;
  } else {
    this->numstages = numstages;
  }
  this->numlevels = this->numstages + 1;
  this->lowest_outlvl = lowest_outlvl;

  for (i=0; i<numlevels; i++) {
    index_a[i] = 0;
    index_d[i] = 0;
  }

  int outlvl = lowest_outlvl;

  // The lowest stage converts from INSAMPLES to OUTSAMPLES
  first_stage = new
    ForwardWaveletStage<SAMPLETYPE, OUTSAMPLE, INSAMPLE>(wavetype,
                                                         rate_l,
                                                         rate_h,
                                                         outlvl,
                                                         outlvl);

  // Setup the remaining stages of the tree (tree represented by
  //  a vector named stages)
  ForwardWaveletStage<SAMPLETYPE, OUTSAMPLE, OUTSAMPLE>* pfws;
  for (i=0; i<this->numstages-1; i++) {
    outlvl++;
    pfws = new ForwardWaveletStage<SAMPLETYPE, OUTSAMPLE, OUTSAMPLE>
            (wavetype, rate_l, rate_h, outlvl, outlvl);
    stages.push_back(pfws);
  }
}
```

In the code example that follows, we show how to set up the reconstruction structure using reverse stages. The constructor code shown takes as input the number of stages in the reconstruction, the wavelet filter type, the upsample rates of the low-pass and high-pass branches and the lowest input level. After argument checks and data member initialization for sample annotation is complete, the last stage is instantiated which converts from an *INSAMPLE* to an *OUTSAMPLE*. Next, the remaining reverse stages are instantiated and represented as a vector of stages. The remaining stages work solely on *OUTSAMPLE* types.

Non-uniform reconstruction using reverse stages

```
template <typename SAMPLETYPE, class OUTSAMPLE, class INSAMPLE>
StaticReverseWaveletTransform<SAMPLETYPE, OUTSAMPLE, INSAMPLE>::
StaticReverseWaveletTransform(const unsigned numstages,
                              const WaveletType wavetype,
                              const unsigned rate_l,
                              const unsigned rate_h,
                              const int lowest_inlvl)
{
  // Argument checks and data initializations
  if ( (numstages == 0) || (numstages > MAX_STAGES) ) {
    this->numstages = 1;
  } else {
    this->numstages = numstages;
  }
  unsigned i;
  this->numlevels = this->numstages+1;
  this->lowest_inlvl = lowest_inlvl;
  this->index = 0;
  this->incoming_index_init = false;
  for (i=0; i<MAX_STAGES+1; i++) {
    incoming_index[i]=0;
  }

  // Set up the input signal buffers
  for (i=0; i<numlevels; i++) {
    SampleBlock<INSAMPLE>* psbis = new SampleBlock<INSAMPLE>();
    insignals.push_back(psbis);
  }

  // Set up the buffers that reside between stages
  for (i=0; i<this->numstages-1; i++) {
    SampleBlock<INSAMPLE>* psbis = new SampleBlock<INSAMPLE>();
    intersignals.push_back(psbis);
  }

  // Instantiate the last stage that converts from INSAMPLE to OUTSAMPLE
  last_stage = new ReverseWaveletStage<SAMPLETYPE, OUTSAMPLE, INSAMPLE>
               (wavetype, rate_l, rate_h);

  // Instantiate the remaining reverse stages
  for (i=0; i<this->numstages-1; i++) {
    ReverseWaveletStage<SAMPLETYPE, INSAMPLE, INSAMPLE>* prws =
      new ReverseWaveletStage<SAMPLETYPE, INSAMPLE, INSAMPLE>(wavetype,
                                                              rate_l,
                                                              rate_h);
    stages.push_back(prws);
  }
}
```

From the two-band wavelet filter banks, a structure that we refer to as stages, many other types of decompositions can be created. A uniform decomposition that looks like a full binary tree can

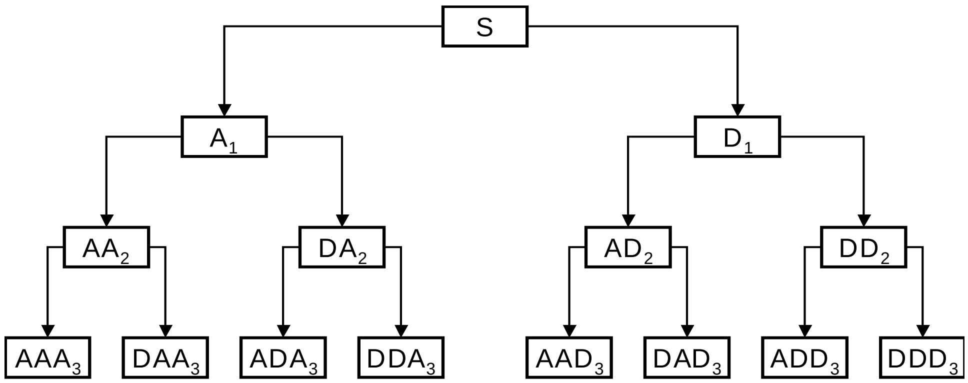


Figure 30: Wavelet packet decomposition tree at level three.

be constructed using the stages by choosing the correct container type for holding the stages (i.e. a list of stages) and devising the correct level annotations. We will discuss uniform decompostitions next in the wavelet packet section.

7.2 Extensions

Based on the flexible design of the Tsunami toolkit, many extensions can be created for different types of purposes. In this section we look at a few of the extensions that are possible with the toolkit. The extensions that we will discuss in this section include wavelet packets, time-varying and adaptive operation, and adding more filter types and coefficients to the library.

7.2.1 Wavelet packets

In what follows from our discussion of filter banks from Section 3, signal decompositions can be non-uniform such as the wavelet decompositions that we have shown thus far, or they can be uniform depending on how the stages are chained together. A uniform signal decomposition using wavelet basis functions, is known as wavelet packets. Wavelet packets have been shown to be useful when looking for a powerful analysis technique that shapes the decomposition based on the signal. Typically the decomposition is determined based on the entropy information of the signal [13]. If nodes of the full-binary tree provide no information (zero entropy), than that node is removed. It is fairly easy to use the Tsunami toolkit to create a wavelet packet decomposition. A diagram of a full wavelet packet system is shown in Figure 30. Instead of just the approximation signal being split, the detail signals are also split to balance the tree. The wavelet transform, based on this figure, consists of all the left half of the tree and the output of block *D1*.

The decomposition consists of creating a tree using *ForwardWaveletStage* and *ReverseWavelet-Stage* in such a way that the tree is balanced and the decomposition uniform. When one decides to build such a decomposition, the output levels will have to be tagged to each stage appropriately. In addition, any of the standard wavelet basis functions used earlier may be used in the wavelet packet decomposition, and perfect reconstruction can still be maintained. Other trees can be created arbitrarily using the forward and reverse stages, and can be adjusted to the various types of signals to be analyzed. Since the stages are parameterized by the up and down sampling rates, the

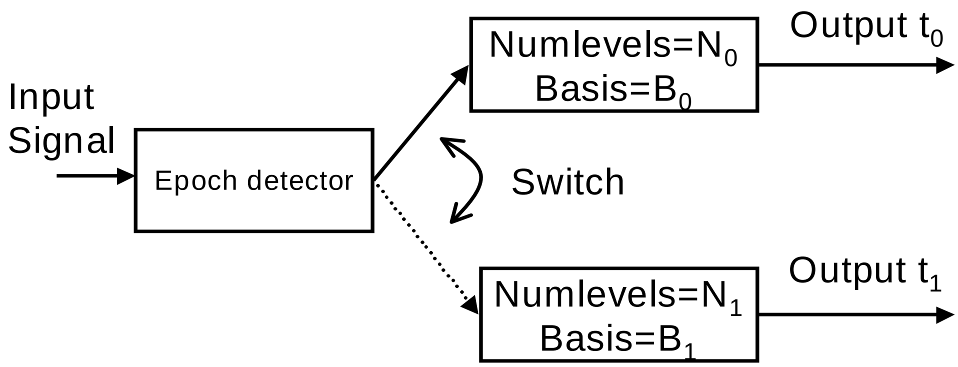


Figure 31: High level view of time-varying operation.

basis function coefficients, and the output level numbers, any type of decomposition is achievable through the correct manipulation of the stage abstraction that we have provided.

In the next subsection, we will talk about how decompositions may adapt to a non-stationary resource signal using time-varying decompositions and through the changing of wavelet basis functions (the filter coefficients) at run-time.

7.2.2 Time-varying, adaptable decompositions

Figure 31 shows a high level view of time varying operation. A detection block peers into the dynamics of the input signal and determines what type of structure and wavelet filters should be used in a particular epoch. The figure shows only two choices but there could be multiple structures and filters used. The Tsunami toolkit offers a dynamic transform class that allows stages to be added and removed dynamically at run-time based on signal detection schemes. In addition, wavelet filter coefficients can also be changed at run-time, and is typically dependent on minimizing computation. This type of operation may prove extremely useful for resource signals that exhibit non-stationary behavior in detectable epochs. In order to properly achieve time-varying operation with low error, it can be decomposed into a detection problem for finding epoch changes followed by transition filters to reduce the error from changing stages and/or coefficients, to the steady state operation once the old structure state has progressed out of the delay line of the filters. The epoch changes may be detected using thresholding, but more sophisticated techniques may need to be employed. This is an area that we will avoid talking about in this report, but is something that we are interested in looking at in more detail.

In order to design transition filters that ease the error of changing the structure of the decomposition at run-time, one should first determine how many different structures will be supported, which translates into the number of different decompositions. The design of the transition filters turn into an optimization problem over the various states between the old and new structure [23, 19]. This is a very powerful and interesting area of work that can be supported by the Tsunami toolkit. In order to support time-varying operation, one would need to add the transition filters to the toolkit, a topic which will be discussed in the next subsection. One would then either build their own decompositions using the forward and reverse stages, or use our class *DynamicForwardWaveletTransform* and *DynamicReverseWaveletTransform*. One limitation of this class is that stages can only be added or removed one at a time. To make this class more powerful, we should offer member functions to change the structure more dramatically, but this can be achieved by calling the member functions *AddStage* and *RemoveStage* a successive number of times.

By looking closely at the transforms that are offered in the toolkit, it appears that the static transform classes also support structure changes. While it is true that the static classes have the ability to change the wavelet filter coefficients and the number of stages at run-time, this removes all stored state in the structure and may cause a prohibitive amount of error between the reconstruction signal and the input signal. This type of operation should be avoided if used in an online system where error needs bounds, and should mainly be used for analysis only.

In the next subsection, a discussion on how to add wavelet filter coefficients and different types of filters is discussed.

7.2.3 Extending the filters

In our base implementation of the Tsunami toolkit, we provide the Daubechies wavelet filters to be used for analysis and synthesis filters. The types that we offer are the Daubechies filters $D2, D4, D6, ..., D20$. Because this set is somewhat limited, there may be a need for more powerful filters in the future. We will now discuss how to add filters to the Tsunami toolkit.

The most simple way to create a new filter is to increase the number of wavelet types and label the new type in the file *waveletinfo.h*. Next, the coefficients need to be added to the file *coefficients.cpp* as well as inserting the new coefficients in the wavelet coefficient table and adding a human-readable name for the filter. This is shown below using the D2 wavelet, the Haar wavelet, as an example.

Filter tables

```
// DAUBECHIES WAVELETS

// N=2, Haar wavelet
const double daub_g2[2] = {1.0/sqrt(2.0),
                           1.0/sqrt(2.0)};

// Add Haar to coefficients table
const double *waveletCoefTable[NUM_WAVELET_TYPES] = {daub_g2,
                                                      daub_g4,
                                                      daub_g6,
                                                      daub_g8,
                                                      daub_g10,
                                                      daub_g12,
                                                      daub_g14,
                                                      daub_g16,
                                                      daub_g18,
                                                      daub_g20};

const unsigned numWaveletCoefTable[NUM_WAVELET_TYPES] = {2,
                                                          4,
                                                          6,
                                                          8,
                                                          10,
                                                          12,
                                                          14,
                                                          16,
                                                          18,
                                                          20};

char *waveletNames[NUM_WAVELET_TYPES] = {"Daubechies 2 (Haar)",
                                         "Daubechies 4",
                                         "Daubechies 6",
                                         "Daubechies 8",
                                         "Daubechies 10",
                                         "Daubechies 12",
                                         "Daubechies 14",
                                         "Daubechies 16",
                                         "Daubechies 18",
                                         "Daubechies 20"};
```

The filters shown in the above table are only the low-pass filter coefficients, $g(n)$. From these coefficients, the high-pass analysis filters and the low-pass/high-pass synthesis filters can be solved for using the CQF properties [22]. If this is not the case, then both the low-pass and high-pass filters for the analysis and synthesis stages will need to be added to the tables appropriately. In addition, the *CQFWaveletCoefficients* class should be avoided, and a new class built that provides the coefficients of the four different filters. Other filter types besides CQF types based on Daubechies' work might be implemented in the future, but is not provided at this time. However, if your filter is of the CQF type, it is very easy to add a new filter to the toolkit, and the code itself generates the other three required filters based on the CQF constraints.

8 System performance and delay

In this section, we detail the performance of the Tsunami toolkit and analyze the real-time system delay of the filter bank structures. The performance tests observe the scalability of adding stages to the decomposition (adds more levels) while keeping the number of samples processed and the wavelet types constant. In addition the scalability of using different wavelet types, corresponding to the filter order, are analyzed while keeping the number of samples to process and the number of stages fixed.

Other performance tests measure CPU overhead as a function of sample rate. The sample rate is sweeped to high rates and measured in terms of measured load and percentage of CPU consumed.

The real-time system delay section analyzes the expected real-time system delay for streaming transforms and discrete transforms. The real-time system delay is an important design constraint for deploying online wavelet-based systems.

8.1 System performance

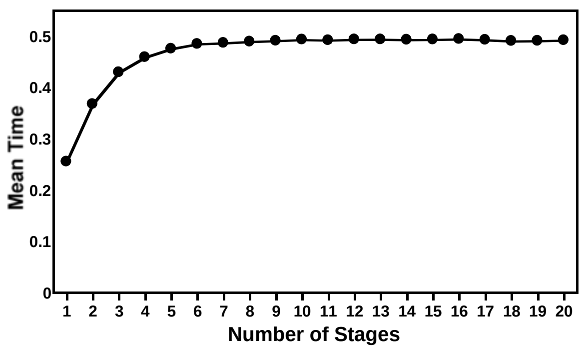
In order to analyze the system performance of the Tsunami toolkit, we have composed several tests to determine the impact of using this system in online distributed applications. The tests are performed using a trace data set of host load sampled at a 1Hz sampling rate [3]. The tests that are run are data independent, but we still use a representative resource signal trace for the performance tests. All tests are run on an unloaded, single processor, 2 GHz Pentium 4 with an 8 KB L1 data cache, and 512 KB L2 cache. The memory size for this machine is 512 MB. The operating system used in the tests is RedHat Linux 7.3, kernel version 2.4.18.

8.1.1 Scalability

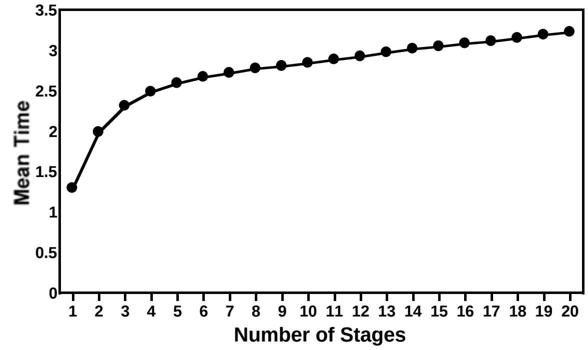
In this section, we measure how the system scales as a function of the parameters listed in Figure 10. Figure 32 (a) and (b) shows the scalability of the streaming forward and reverse transforms as the number of stages are increased from one to twenty while keeping the wavelet type and number of samples processed constant. The wavelet type used in these tests are the D10 and 262,144 samples are processed. The performance metric used in these tests is the mean-time to completion to process all of the input samples. As expected, as the number of stages are increased, the mean completion time tends to level out. This is because as a new stage is added, it must process half the amount of samples as the stage before, and therefore the lessened amount of work tends to flatten the mean completion time. Early stages in the chain perform most of the work in the transform due to having to process the most samples.

In Figure 32 (c) and (d), we show the scalability of the streaming transforms as the wavelet type is increased from D2 (Haar) to D20 while keeping the number of stages and number of samples processed constant. The number of stages used in these tests are 10 and the same number of samples are processed as in (a) and (b). As the filter changes from a D_N to D_{N+2}, the number of additional operations that must be performed is two extra multiplications and accumulations per stage. Therefore, we expect a linear relationship in the mean-time to completion as the wavelet type is increased.

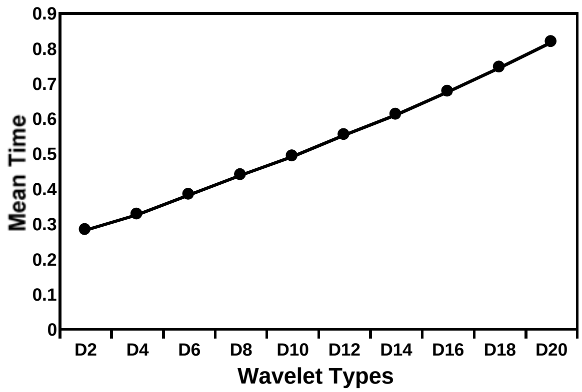
[3] Host load traces are available at http://www.cs.northwestern.edu/∼pdinda/LoadTraces



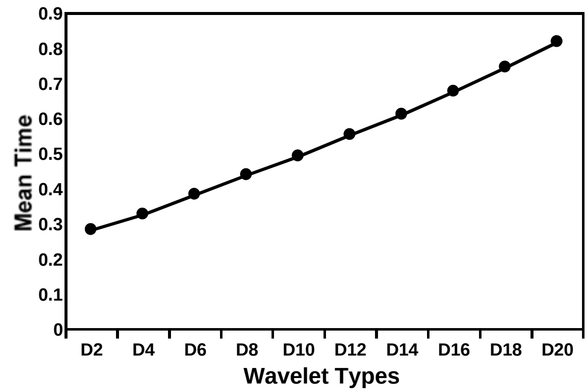
(a)



(b)



(c)



(d)

Figure 32: Streaming transform scalability. Scalability of the (a) forward transform and (b) the reverse transform as stages are added. The wavelet type is a DAUB10 for these tests. Scalability of the (c) forward transform and (d) the reverse transform as the wavelet type increases from DAUB2 to DAUB20. The stages are fixed at 10 for these tests.

Figure 33 (a) and (b) shows the scalability of the discrete forward and reverse transforms as the blocksize is increased from two to 1024 samples while keeping the wavelet type and total number of samples to process constant. The wavelet type used is a D10 and 262,144 samples are processed. The 262,144 samples are split into blocks and the discrete transforms are run successively for each blocksize. The performance metric in this set of tests is again the mean-time to completion. The blocksize in the discrete transforms determine the number of levels in the decomposition. As expected, as the blocksize increases the mean-time to completion decreases. The reason for this is twofold. First, as the blocksize is increased, there are less calls to the discrete transform routine. Secondly, the amount of work to be performed at higher levels, where levels is a function of blocksize, decreases exponentially.

In Figure 33 (c) and (d), we show the scalability of the discrete forward and reverse transforms as the wavelet type is increased from D2 to D20. The blocksize for these tests are 1,024 samples and 262,144 samples are processed to completion. For reasons discussed above, the relationship of wavelet type to mean completion time is linear as the wavelet type is increased. This is again due to the constant increase in the number of operations performed for each new wavelet type at each stage.

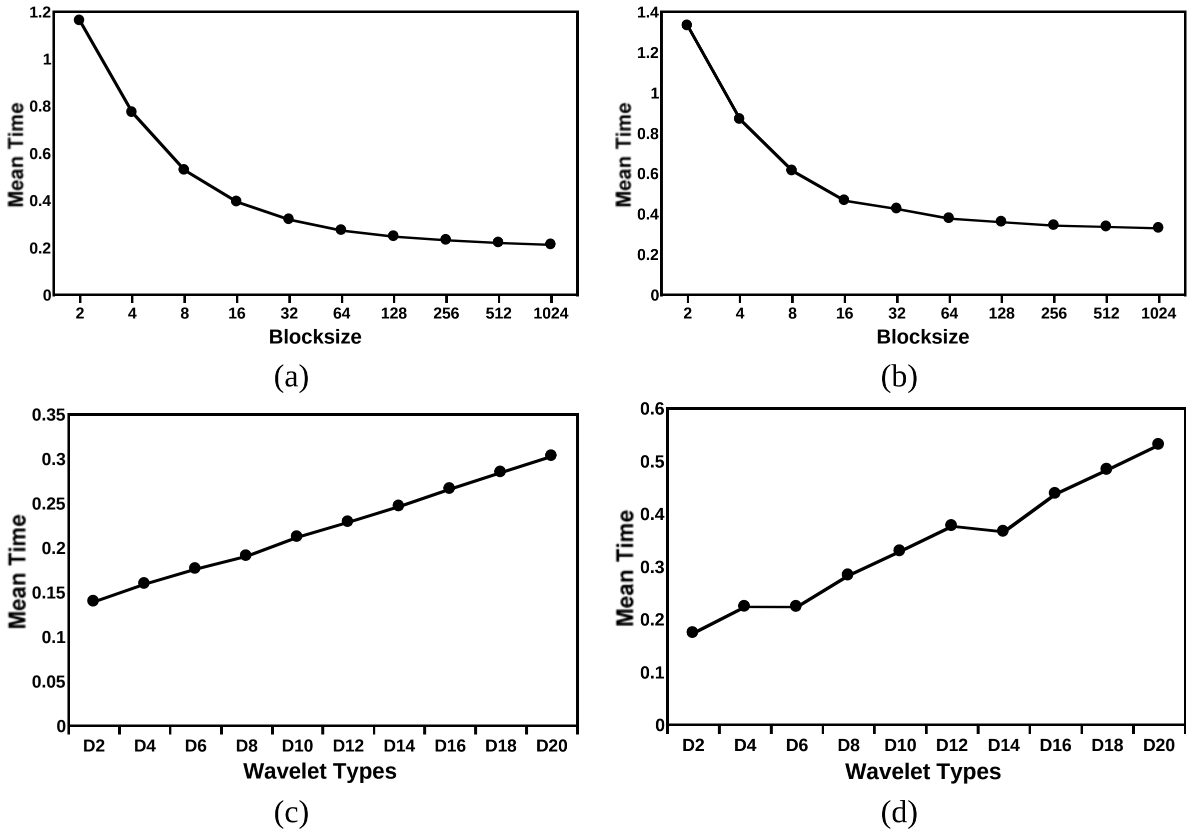


Figure 33: Discrete transform scalability. Scalability of the (a) discrete forward transform and (b) the discrete reverse transform as the blocksize varies from 2 to 1024. The wavelet type is a DAUB10 for these tests. Scalability of the (c) discrete forward transform and (d) discrete reverse transform as the wavelet type changes from DAUB2 to DAUB20. The blocksize is fixed at 1024 for these tests.

8.1.2 Performance as a function of sample rate

In this section, we measure how the system performs as a function of the sample rate. All tests in this section first start a vmstat monitor to measure the percentage of cpu consumed over time. After a quiesce time of 50 seconds, a loadmonitor is started to estimate the measured load on the machine. At the beginning of the test, a rather large data file is loaded and the system quiesces for another 50 seconds. After this period is over, we sweep the sampling rate from 5.12 kHz to 327.8 kHz followed by a maximum rate test where 1024 blocks of size 65,536 samples are run as fast as possible. After the max rate test, the system returns for 400 seconds to a state where just the vmstat and the load monitor are running.

Figure 34 shows the performance of streaming transforms as a function of sample rate. In (a), the percentage CPU used is shown for the streaming forward transform. The system can sustain a sample rate on the order of 40 kHz while keeping the percentage of CPU used under 10%. In (b), the percentage CPU used is shown for the streaming reverse transform. The reverse transform performs a bit worse than the forward transform. This is probably due to the extra additions per stage as realized in the reverse transform. The reverse streaming transform also sustains a sample

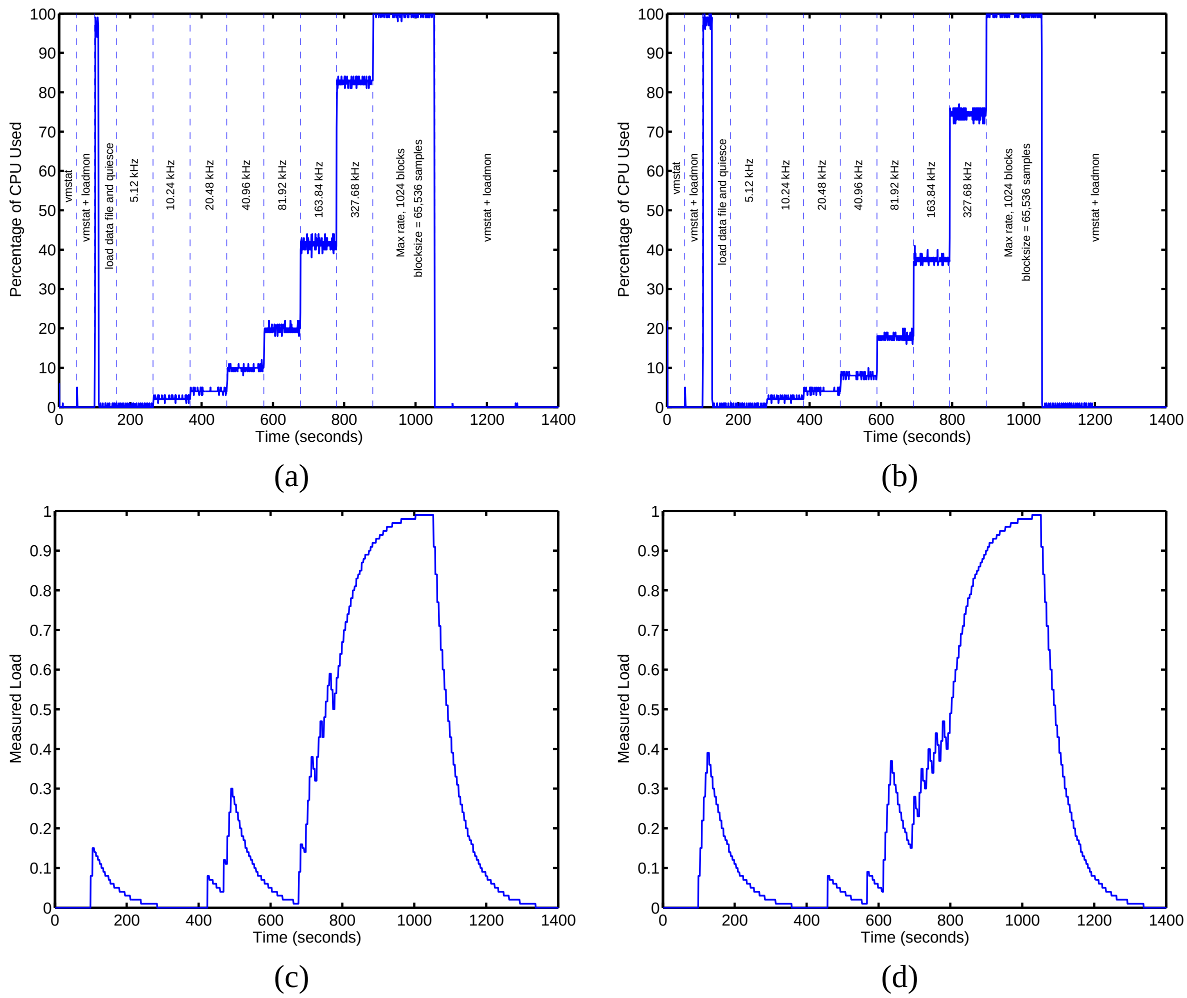


Figure 34: Streaming transform performance as a function of sample rate. Percentage of CPU used as a function of sample rate for the (a) forward and (b) reverse transform. Measured load as a function of sample rate for the (c) forward and (d) reverse transform.

rate on the order of 40 kHz while keeping the percentage of CPU used under 10%. In (c) and (d) the measured load as a function of time and sample rate are shown for the forward and reverse transform respectively. The measured load is typically under 0.2 for sample rates as high as 40 kHz. For typical sample rates used in distrubuted systems (i.e. 1 Hz), the load and percentage of CPU used is negligible. This can be inferred from the load and percent CPU used at 5.12 kHz. At this rate, the percent CPU used is in the noise and is commonly 0 or 1 percent. Due to the averaging nature of load measurements in the linux operating system, it is hard to estimate the measured load at a sampling rate of 5.12 kHz rate, but the measured load at 10.24 kHz is zero, so it can also be inferred that the load is zero for a 5 kHz rate.

Figure 35 shows the performance of the discrete transforms as a function of sample rate and time. In (a) and (b) we show the percentage of CPU consumed as a function of sample rate for the discrete forward and discrete reverse transforms respectively. The discrete transforms perform better as a function of sample rate when compared with the streaming transforms. Both transform

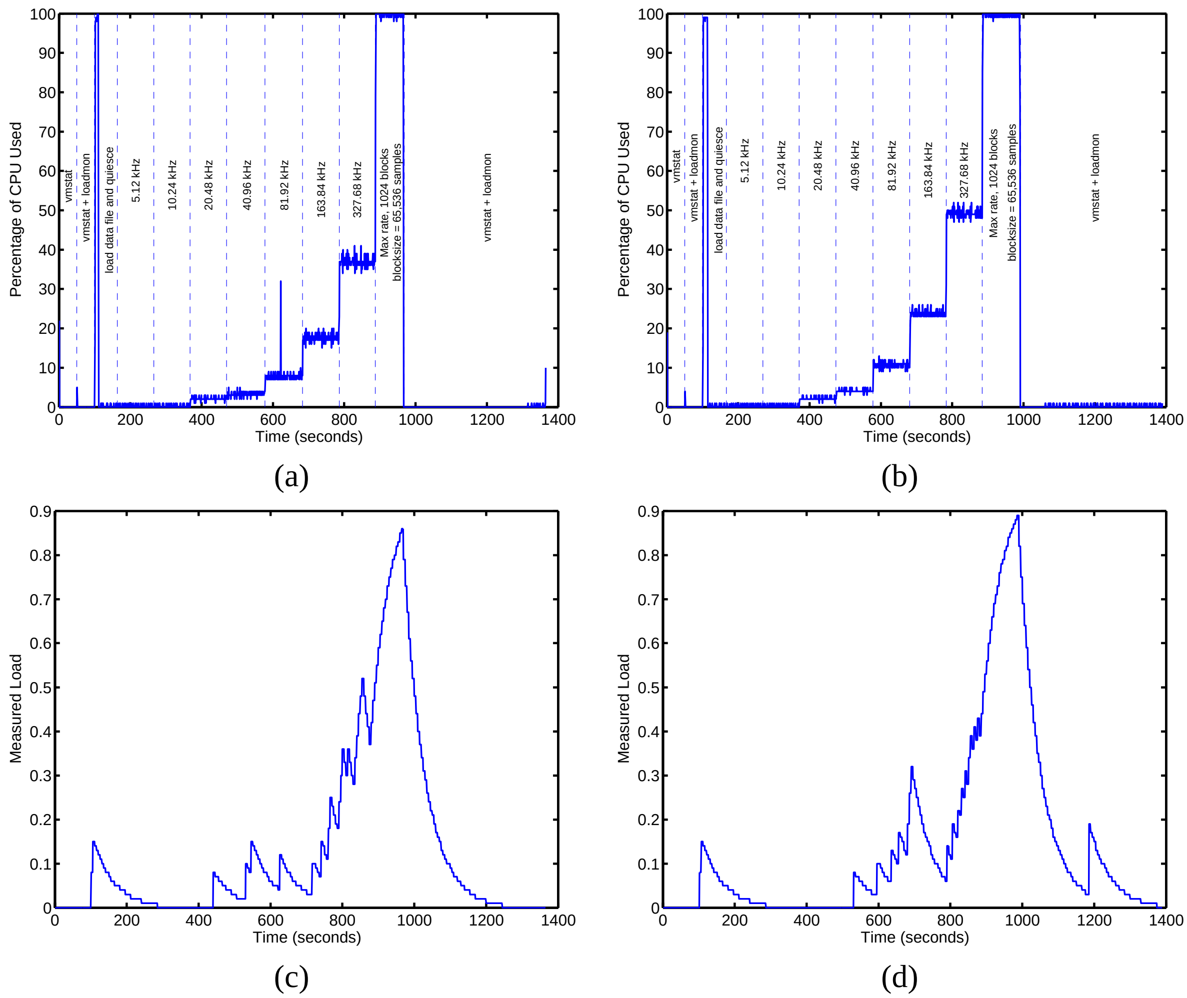


Figure 35: Discrete transform performance as a function of sample rate. Percentage of CPU used as a function of sample rate for the (a) forward and (b) reverse discrete transforms. Measured load as a function of sample rate for the (c) forward and (d) reverse discrete transforms.

Transform type	Samples per second
Streaming forward transforms	$156.10K$
Streaming reverse transforms	$131.84K$
Discrete forward transforms	$185.99K$
Discrete reverse transforms	$177.02K$

Figure 36: Average maximum samples per second sustained for streaming and discrete transforms.

directions can sustain a sample rate of 81.92 kHz while keeping the percentage of CPU used less than 14%. The measured load is also less than 0.2 for sample rates at or around 81.92 kHz. This is to be expected since the discrete transforms are based on an efficient algorithm [12].

Figure 36 shows the average maximum sampling rate achievable by the various transform methods. In this test each of the various transform methods listed are run as fast as possible. The number of samples processed in this test are 8,388,608 samples. The time is measured and the number of

Transform type	System delay, n_d
Streaming transforms	$2^M(N-1)+2-N+2\cdot Ts_{COMP}$
Discrete transforms	$2^M \cdot \Delta T + 2\cdot Td_{COMP}$

Figure 37: Real-time system delay for each type of transform.

samples per second are calculated. The reverse transforms perform a bit worse than the forward transforms, but this is to be expected due to extra operations.

8.2 Real-time system delay

The types of transforms that are provided in the base implementation of the Tsunami toolkit, have varying real-time system delays based on the type of transform and the wavelet filter type used. The real-time system delays for each transform type is shown in Figure 37. In the figure, M designates the number of stages ($M+1$ is the number of levels), N is the number of filter coefficients for a particular wavelet type ($N=2$ for the Haar wavelet), ΔT is the sampling period, and Ts_{COMP} and Td_{COMP} is an estimate of the computational time for performing the streaming or discrete transform respectively.

As can be seen in the table, each type of transform contains exponential real-time system delays in terms of the number of stages in the decomposition. For low sampling rates using a large number of decomposition levels, this delay may be prohibitive. Finding ways to minimize real-time system delay is an area of research that we are actively pursuing.

9 Interface to RPS

Tsunami fits into the RPS system as a package whose description and interface are as described above. However, there are three additions. First, there is a set of interface classes that define wavelet information that can be easily serialized over a communication channel. Second, a Wavelet prediction model has been added to the RPS TimeSeries module. Third, there are a set of RPS components, utility programs, that implement various operations. Among these components are predictors that can use the Wavelet model. The interface classes enable the easy construction of the components. The components can be composed at run-time to create different kinds of wavelet systems that communicate over the network. The interface classes are not Tsunami-specific, although their implementations, and the implementation of some of the components, are. This means that they can potentially be integrated with other tools.

9.1 Interface classes and types

RPS's communication model is designed to support streams of C++ objects, and request-response operation, in which a request C++ object is sent to a server which then returns a response C++ object, a simple form of synchronous RPC. All objects are serialized to a machine-independent binary format. Each serializable class implements an interface called *SerializeableInfo* and defines methods for packing and unpacking its data. Generally, each class contains all the context needed to interpret its contents.

Type	Description
Type information	
WaveletType	Underlying wavelet (e.g., Daubauchies 8)
WaveletRepresentationType	Domain (time, frequency, wavelet approx/detail/both)
WaveletBlockEncodingType	Ordering of data in the block (pre-, in-, post-order)
WaveletRepresentationInfo	All metadata needed to use a sample Contains WaveletType, WaveletRepresentationType number of levels, and sampling period
WaveletTransformDirection	Direction of transform (forward, reverse)
WaveletTransformRequestType	Full specifies a transform except for data Contains WaveletTransformDirection, to and from WaveletRepresentationInfo, and to/from WaveletBlockEncodingType
Sample blocks	
WaveletBlock	Self-contained, timestamped block of samples Contains WaveletRepresentationInfo, WaveletBlockEncodingType, number of samples, and array of doubles.
Samples and sample blocks for streaming	
WaveletIndividualSample	Timestamped sample with all necessary metadata Contains WaveletReprsentationInfo, index, level timestamp, tag, and value
WaveletStreamingBlock	Timestamped block of samples with all necessary metadata Contains timestamp, tag, number of samples, and array of WaveletIndividualSamples.
Discrete transforms	
WaveletTransformBlockRequest	A discrete transform request Contains WaveletTransformRequestType, WaveletBlock, tag, and input and output timestamps
WaveletTransformBlockResponse	A discrete transform response Identical to WaveletTransformBlockRequest
Streaming transforms	
WaveletTransformRequestType	Specifies type of transform (see above)
WaveletIndividualSample	Stream content (see above)
WaveletStreamingBlock	Stream content (see above)

Figure 38: RPS interface classes and types

Figure 38 summarizes the classes and types involved in the interface to RPS. Each item in the list supports serialization to a lightweight binary format. RPS's communication template library uses this interface to send data over different channels, such as TCP connections, UDP streams, and others. The interface supports both streaming and block transforms. Every request, response, sample, or sample block has associated with it all the necessary contextual information to make sense of it. While wavelet transforms are not stateless, to the greatest extent possible the interface attempts to push state information into data that is communicated over the network.

Each sample or coefficient, which is a double precision floating point value is either decorated with contextual information or is contained in a block that contains this information. For example, a *WaveletBlock* is a self-contained block of samples (an array of doubles) that also contains a timestamp, a sampling period, a *WaveletRepresentationInfo*, and *WaveletBlockEncodingType*. The *WaveletRepresentationInfo* includes a *WaveletRepresentationType*, which tells us whether the sample block is in time, frequency, or wavelet domain. In wavelet domain, the numbers may represent the detail signals, the approximation signals, or both. For wavelet domain, the *WaveletRepresentationInfo* also tells us how many levels are being used. The *WaveletRepresentationInfo* also includes the *WaveletType* (underlying wavelet used). The *WaveletBlockEncodingType* describes whether the block is in pre-, in-, or post-order traversal form, if the block is in wavelet domain.

WaveletBlocks are used for one-off discrete wavelet transforms. For streaming operation, RPS includes a *WaveletIndividualSample* and a *WaveletStreamingBlock*. A *WaveletIndividualSample* contains a single timestamped value, its index and level, and a *WaveletRepresentationInfo* which describes the context of the value as above. A *WaveletStreamingBlock* is an array of *WaveletIndividualSamples*.

WaveletTransformRequestType describes the wavelet transform to be done. It contains a *WaveletTransformDirection*, stating whether a forward or inverse transform is needed, and a (*WaveletRepresentationInfo*, *WaveletBlockEncodingType*) pair for both the input and output data. Combined with data, this fully specifies a transform to be done.

To accomplish a discrete transform on a block, one constructs a *WaveletTransformBlockRequest*, which contains the *WaveletTransformRequestType* and the data, sends it to the server, and receives back a *WaveletTransformBlockResponse*, which contains the transformed data and a *WaveletTransformRequestType* that explains exactly what was done.

For streaming transforms, a *WaveletTransformRequestType* is used to specify the transform (no *WaveletBlockEncodingType* is used). The transform than outputs a stream of *WaveletIndividualSamples* or *WaveletStreamingBlocks*. The stream contents provide all information necessary to interpret the transformed data.

9.2 Wavelet predictor

The RPS TimeSeries module has been extended to include a wavelet prediction model that can be used from any TimeSeries-based tool. The basic idea behind the wavelet predictor is to transform an incoming signal into wavelet detail signals. A non-wavelet predictor (or delay component) is then run on each level separately, and the detail predictions are then inverse transformed to get the predicted signal.

The predictor specification is `WAVELET file`, where `file` is a configuration file, which has the following format:

Wavelet prediction configuration file format

```
# Comment
# Number of levels
3
# Type of wavelet (D8 here)
3
# For each level: level predhorizon model|delay
0 +1 managed 50 50 30 0.01 0.01 ar 16
1 +8 managed 50 50 30 0.01 0.01 ar 16
2 +8 managed 50 50 30 0.01 0.01 ar 16
```

Instead of a predictor, a delay may also be used, denoted `delay` and having a negative "prediction horizon". We provide a script, *generate_wavelet_prediction_config.pl* to help in producing such configuration files.

There is a significant caveat with the current implementation of wavelet prediction. The configuration file specifies a structure that generates a single output signal, for k steps ahead or behind realtime depending on the configuration file. The RPS predictor model, however, allows the user to ask for prediction for any number of steps into the future. In the current implementation, the k-ahead output value is always reported. This bug will be fixed in a future version of the predictor.

9.3 Wavelet components

Using the interface classes and types, we built several RPS prediction components. Components in RPS are simple programs that can be tied together at run-time to build different kinds of systems. They provide a way of using RPS without writing any code. Sophisticated users can create their own components.

Figure 39 summarizes the wavelet components that are included. There are three categories of components: discrete transforms, streaming transforms, and multiresolution queries.

Discrete transforms are straightforward. There is a stateless server that accepts requests for transformations. The client packages up a request along with sample data, sends it to the server, and the server replies with the transformed data.

In streaming transforms, streams of RPS *Measurements* are transformed into streams of *WaveletIndividualSamples* or *WaveletSampleBlocks*, and conveyed over the network to a client that can display, filter, or reconstruct from them. The majority of the work is done in *wavelet_streaming_server*, which accomplishes the transform, and *wavelet_streaming_client*, which displays or reconstructs. In addition, the transformed data can be buffered using *wavelet_buffer*, and retrieved using the utility *wavelet_buffer_client*. This provides convenient request-response access to the buffered stream.

Multiresolution queries build on top of streaming transforms. The *wavelet_streaming_selection* component can be configured to let only samples within a range of levels pass. The component *Wavelet_streaming_query* can reconstruct the *Measurement* stream using only a subset of the available levels. *Wavelet_interval_query* is similar, except it computes an average over an interval of time. *Wavelet_streaming_denoise* is a filter similar to *wavelet_streaming_query*, except that it discards *WaveletIndividualSamples* not based on their level, but on their energy.

We have begun to build prediction tools using wavelets. The *wavelet_predict* component can be used to project forward wavelet detail signals, streams of *WaveletStreamingSamples*. The goal of

Component	Description
Block Transforms	
wavelet_reqresp_server	One-off server for block transforms
wavelet_reqresp_client	one-off client for block transforms
Streaming Transforms	
wavelet_streaming_server	Transforms a stream of Measurements into a stream of WaveletIndividualSamples
wavelet_streaming_client	Reads a stream of WaveletIndividualSamples and either prints them or reconstructs the original signal
wavelet_buffer	Buffers a stream of WaveletIndividualSamples and provides request/response access to it
wavelet_bufferclient	Requests WaveletIndividualSamples from a wavelet_bufferclient
Wavelet-based multiresolution queries	
wavelet_streaming_server	As above
wavelet_streaming_selection	Reads a stream of WaveletIndividualSamples and emits a stream that contains only the specified levels
wavelet_streaming_denoise	Reads a stream of WaveletIndividualSamples and emits a stream that contains only values greater than a specified limit
wavelet_streaming_query	Reconstruct from multicasted streams
wavelet_interval_query	Reconstruct average over interval from multicasted streams
Wavelet-based prediction	
wavelet_predict	Reads a stream of WaveletIndividualSamples and emits a stream of WaveletIndividualSamples projected into the future using models specified in a configuration file. Used to explore prediction as a cure for the delay problem.
predserver managed_predserver	These are the standard RPS prediction servers. Each reads a stream of Measurements and produces a stream of Predictions. A WAVELET predictive model is now supported. Measurements are wavelet-transformed, prediction is done on each level and the predictions are superposed to get final output.

Figure 39: RPS wavelet components

wavelet_predict is to minimize the real-time system using prediction. The *wavelet_predserver* component attempts time-series prediction by first wavelet-transforming an incoming measurement stream, predicting each level using a separate prediction filter, and then combining the predictions at the output.

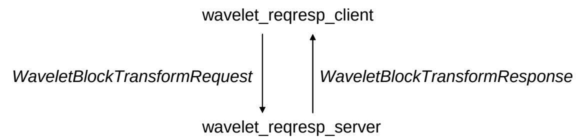


Figure 40: Request/Response configuration.

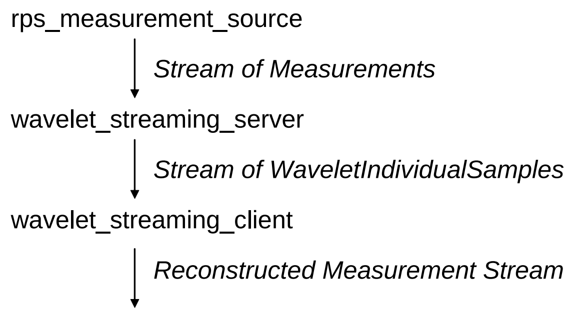


Figure 41: Streaming client/server configuration.

9.4 Configurations

The wavelet components can be composed to create various sorts of systems. We have experimented with several configurations.

Figure 40 illustrates a simple request/response configuration. One *wavelet_reqresp_server* can provide discrete wavelet transform services for the network.

Figure 41 illustrates simple streaming operation. We acquire measurements from some RPS sensor (host load, network bandwidth, etc). A *wavelet_streaming_server* transforms these into *WaveletIndividualSamples.* A *wavelet_streaming_client* can then connect to this stream and either print it directly or reconstruct the original measurement stream from it.

Figure 42 shows a generalization of this, providing multiresolution queries. Here, the stream from *wavelet_streaming_server* is acquired by multiple *wavelet_streaming_selection* components. Each emits a subrange of the levels in the original stream to a different ip multicast channel. The *wavelet_streaming_query* component connects to only those channels needed to reconstruct the

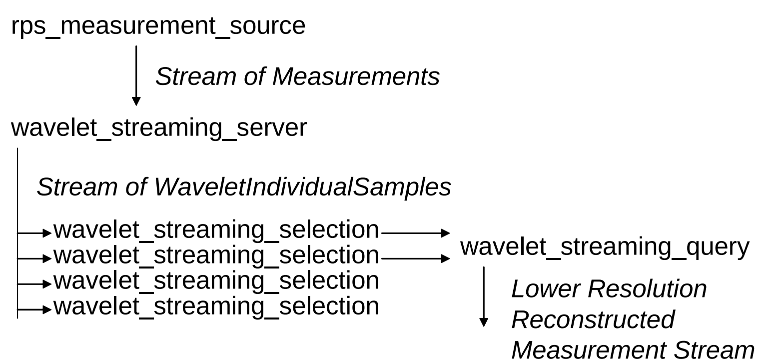


Figure 42: Multiresolution streaming configuration.

signal to the resolution needed. The network traffic is determined by the number of levels needed by the *wavelet_streaming_query* component of maximum resolution.

10 Conclusions and future work

Wavelet techniques have been shown useful in analyzing computer generated resource signals such as network bandwidth, host load, and IP flow data. An emergence of wavelet-based online systems have been deployed in the literature for estimating network problems, but many of these solutions are ad hoc. In order to address our research needs, we have built a general and extensible wavelet-based system that can be used for offline analysis and online system building. This system can be used for many areas related to our research goals since the system provides building blocks for general decompositions, time-varying operation and streaming modes of operation which we feel will be important to distributed system research. The system provides standard interfaces such as the discrete wavelet transform and its inverse as well as an extensive MRA analysis interface. From the interfaces provided in the toolkit, flexibility is in the hands of the researcher for progressing through simulation to deployment of an actual online system. The toolkit performs well at high sampling rates, rates much higher than we typically observe in measurement sensors in distributed systems, and scales well as the wavelet type or number of stages are increased. Tsunami fits snugly into the RPS toolkit, and uses its interfaces for communication, resource monitoring and prediction.

Future directions of our research include obtaining a better understanding of computer generated resource signals in order to predict the behavior of applications that run in distributed systems. Wavelet approaches have already been shown useful for understanding and visualizing complex signals like those found in computer measurement systems. Combinations of dimensionality reduction techniques of multivariate resource signals with that of a thorough wavelet analysis may lead to a better understanding of these signals, and therefore, an enhancement of the predictability of application run-time signatures.

Other directions include novel approaches to minimizing the real-time system delay incurred while using wavelet transform techniques. An application with stringent, real-time delay constraints may find wavelet-enabled techniques prohibitive for use in their online application. We have created a resource dissemination system using wavelet techniques to summarize and disseminate information efficiently throughout the network. The system decouples sensors which measure resources and the applications of various granularities which subscribe to measurements. However, the real-time system delay prohibits the use of this system with fine-grain, interactive applications. Solving this problem lends more flexibility in building online wavelet systems, and provides a more general solution to many domains. The system delay problem has no effect on offline analysis.

We are in the process of looking for other applications that may benefit from wavelet techniques. Among these include using wavelets for signature detection. Applications of signature detection include intrusion detection on hosts or anamoly detection in segments of the network. These areas are new approaches for us, and we feel that our research will benefit from the toolkit that we have built.

Appendix

A Tsunami class descriptions

In this section of the appendix, we list the member functions and data members of each class with a brief description of the functionality of each. This section is for quick reference when using the Tsunami toolkit.

A.1 Command line functions

We have built many functions for making the command line utilities more readable and also for factoring code with common functionality. In order to use these functions and typedefs, the file *cmdlinefuncs.h* must be properly included. The typedefs that are used in the utilities and the function names and descriptions of each are listed here.

Type defines

```
typedef WaveletInputSample<double> wisd;
typedef WaveletOutputSample<double> wosd;
```

Function name	Description
WaveletType GetWaveletType(const char *x, const char *filename);	Obtains the type of wavelet from char string.
void ParseSignalSpec(SignalSpec &spec, ifstream &file);	Parses the signal specification.
void ParseZeroSpec(vector⟨int⟩ &spec, ifstream &file);	Parses the zero specification.
void OutputWaveletCoefs(ostream &os, vector⟨vector⟨wosd⟩ ⟩ &levels);	Outputs the wavelet coefficents in standard output form.
void OutputWaveletCoefs (ostream &os, vector⟨WaveletOutputSampleBlock⟨wosd⟩ ⟩ &levels, const TransformType tt);	Same as above but takes blocks of samples instead of samples. Uses the transform type to print the appropriate number of levels.
unsigned OutputWaveletCoefs (ostream &os, vector⟨WaveletOutputSampleBlock⟨wosd⟩ ⟩ &levels, const TransformType tt, const unsigned start_index);	Same as above but outputs data starting at the specified index.
void OutputWaveletCoefs (ostream &os, const DiscreteWaveletOutputSampleBlock⟨wosd⟩ &dwosb, const TransformType tt, const bool flat);	Same as above but works on discrete output. Takes flat as input to generate human readable output as well.
void OutputMRACoefs(ostream &os, vector⟨vector⟨wosd⟩ ⟩ &approxlevels, vector⟨vector⟨wosd⟩ ⟩ &detaillevels);	Outputs the coefficients of an MRA analysis. Tags each output line with the appropriate signal type.
void OutputMRACoefs (ostream &os, vector⟨WaveletOutputSampleBlock⟨wosd⟩ ⟩ &approx, vector⟨WaveletOutputSampleBlock⟨wosd⟩ ⟩ &detail);	Same as above but works on output sample blocks.
unsigned OutputMRACoefs (ostream &os, vector⟨WaveletOutputSampleBlock⟨wosd⟩ ⟩ &approx, vector⟨WaveletOutputSampleBlock⟨wosd⟩ ⟩ &detail, const unsigned index);	Same as above but outputs data starting at the specified index.
void OutputLevelMetaData(ostream &os, vector⟨vector⟨wosd⟩ ⟩ &levels, const unsigned numlevels);	Outputs the sizes of each wavelet coefficient level.
void OutputLevelMetaData (ostream &os, vector⟨WaveletOutputSampleBlock⟨wosd⟩ ⟩ &levels, const unsigned numlevels);	Same as above but works on output sample blocks.
void OutputLevelMetaData(ostream &os, const unsigned *levelsize, const unsigned levelcnt);	Same as above but works on arrays and counts.
void OutputLevelMetaData (ostream &os, const DiscreteWaveletOutputSampleBlock⟨wosd⟩ &dwosb, const TransformType tt);	Same as above but works on discrete output. Used transform type to infer the number of levels in the representation.

A.2 Flat parsing

Much of the output from the command line utilities are in a parsing format that is difficult to understand using human eyes. Therefore, we have provided *FlatParser*, a class that parses flat output from the command line utilities. Here we show the member functions included in this class.

Member function	Description
FlatParser();	Default constructor.
virtual ∼FlatParser();	Destructor.
void ParseTimeDomain(vector⟨wisd⟩ &samples, istream &in);	Parses time-domain samples, expecting one sample per line.
void ParseTimeDomain(deque⟨wisd⟩ &samples, istream &in);	Same as above but output stored to deque instead of vector.
bool ParseWaveletCoefsSample(vector⟨wosd⟩ &wavecoefs, istream &in);	Parse wavelet coefficients at sample times into a vector.
void ParseWaveletCoefsBlock (vector⟨WaveletOutputSampleBlock⟨wosd⟩ ⟩ &wavecoefs, istream &in);	Same as above except that the coefficients are stored in a vector of blocks.
void ParseWaveletCoefsBlock (DiscreteWaveletOutputSampleBlock⟨wosd⟩ &wavecoefs, istream &in);	Same as above except that the coefficients are packed into a discrete block.
unsigned ParseWaveletCoefsBlock (vector⟨WaveletOutputSampleBlock⟨wosd⟩ ⟩ &wavecoefs, istream &in, const unsigned parsenum);	Same as above except that a number of coefficients are parsed each call. This is used for the dynamic transforms.
bool ParseMRACoefsSample (const SignalSpec &spec, vector⟨wosd⟩ &acoefs, vector⟨wosd⟩ &dcoefs, istream &in);	Parses MRA coefficients at sample times into two separate vectors, one for approximations and one for details. The returned vectors are based upon the signal specification.
bool ParseMRACoefsSample(vector⟨wosd⟩ &acoefs, vector⟨wosd⟩ &dcoefs, istream &in);	Same as above except that all the available approximations and details are returned.
void ParseMRACoefsBlock (const SignalSpec &spec, vector⟨WaveletOutputSampleBlock⟨wosd⟩ ⟩ &acoefs, vector⟨WaveletOutputSampleBlock⟨wosd⟩ ⟩ &dcoefs, istream &in);	Parses MRA coefficients all at once and stores them into blocks. The returned vectors of blocks are based upon the signal specification.
unsigned ParseMRACoefsBlock (const SignalSpec &spec, vector⟨WaveletOutputSampleBlock⟨wosd⟩ ⟩ &acoefs, vector⟨WaveletOutputSampleBlock⟨wosd⟩ ⟩ &dcoefs, istream &in, const unsigned parsenum);	Same as above except that a number of coefficients are parsed each call. This call is used for the dynamic transforms.
void ParseMRACoefsBlock (vector⟨WaveletOutputSampleBlock⟨wosd⟩ ⟩ &acoefs, vector⟨WaveletOutputSampleBlock⟨wosd⟩ ⟩ &dcoefs, istream &in);	Same as above except that all the available approximations and details are returned.

A.3 Wavelet information

This set of data types is for adding new filters, and parameters within the Tsunami toolkit. If one is to add a new wavelet type, they must edit the file *waveletinfo.h* within the *include* directory of the wavelet branch.

Data type	Description
const int NUM_WAVELET_TYPES;	The number of different types of basis functions supported.
enum TransformType;	Then types of transforms that are supported by the toolkit. These include TRANSFORM, APPROX and DETAIL.
enum WaveletType;	An enumeration for each wavelet type. An example is DAUB2, which is simply the Haar basis function.
const numsigned numberOfCoefs[NUM_WAVELET_TYPES];	For each wavelet type, designate the number of coefficients.
const unsigned MAX_STAGES;	The maximum number of stages that can be chained together. This number is currently set to 20, which implies a maximum decomposition of 21 levels.

A.4 Sample classes

The purpose of the sample classes is to provide a generically typed class that is tagged with the index and value of a sample.

A.4.1 Sample base class

This class contains most of the operations for manipulating the data members of the sample classes. In the next two tables, we provide the data members and member functions of the *Sample* class.

Data member	Description
Protected	
SAMPLETYPE value;	The Sample base class is parameterized by the typename SAMPLETYPE. This can be any of the machine dependent datatypes such as int, double, etc. The value holds the sample's value and must be of type SAMPLETYPE.
unsigned index;	Because we expect samples to be periodic, we only need to keep an index number for maintaining sample order. This data type is for maintaining sample order.

Member function	Description
Public	
Sample(const SAMPLETYPE value=0, const unsigned index=0);	Default constructor that takes as arguments the sample value and index.
inline Sample(const Sample &rhs);	Copy constructor.
virtual ∼Sample();	Destructor.
virtual Sample⟨SAMPLETYPE⟩ & operator=(const Sample &rhs);	Equal operator that takes as argument a reference to another Sample and returns a reference to the copied Sample.
Sample⟨SAMPLETYPE⟩ & operator=(const SAMPLETYPE rhs);	Equal operator that takes as argument a value and returns a reference to the copied Sample.
Sample⟨SAMPLETYPE⟩ & operator+(const SAMPLETYPE rhs);	Plus operator that takes as input argument a value and performs value = value + rhs. It returns a reference to the Sample.
Sample⟨SAMPLETYPE⟩ & operator+(const Sample &rhs);	Plus operator that takes as input a reference to a Sample and performs value = value + rhs.value. It returns a reference to the Sample.
Sample⟨SAMPLETYPE⟩ & operator+=(const SAMPLETYPE rhs);	Operator that takes as input a sample value and performs value = value + rhs. It returns a reference to the Sample.
Sample⟨SAMPLETYPE⟩ & operator+=(const Sample &rhs);	Operator that takes as input a reference to a Sample and performs value = value + rhs.value. It returns a reference to the Sample.
SAMPLETYPE operator*(const double rhs);	Operator that takes as input the type double and returns a SAMPLETYPE of the result value*rhs.
inline void SetSampleValue(const SAMPLETYPE sample);	Sets the sample value.
inline SAMPLETYPE GetSampleValue();	Gets the sample value.
virtual inline void SetSampleIndex(const unsigned index);	Sets the sample index.
virtual inline unsigned GetSampleIndex() const;	Gets the sample index.
virtual ostream & Print(ostream &os) const;	Prints the Sample data.
virtual ostream & operator≪(ostream &os) const;	Operator for printing the Sample data.

A.4.2 InputSample class

This class is simply an intermediate class that sits between the base class, *Sample*, and specialized input sample classes which derive from it. It has no data members and very few member functions.

Member function	Description
Public	
InputSample(const SAMPLETYPE value=0, const unsigned index=0);	Default constructor that takes as input the value and index of the sample.
InputSample(const InputSample &rhs);	Copy constructor that takes as input a reference to an InputSample.
virtual ∼InputSample();	Destructor.
virtual InputSample⟨SAMPLETYPE⟩ & operator=(const Sample⟨SAMPLETYPE⟩ &rhs);	Equal operator that takes as input a reference to a Sample and returns a reference to an InputSample.

A.4.3 OutputSample class

This class is an intermediate class between the base class, *Sample*, and specialized output sample classes which derive from it. It has no data members and very few member functions.

Member function	Description
Public	
OutputSample(const SAMPLETYPE value=0, const unsigned index=0);	Default constructor that takes as input the sample value and index.
OutputSample(const OutputSample &rhs);	Copy constructor that takes as input a reference to an OutputSample.
virtual ∼OutputSample();	The destructor.

A.4.4 WaveletInputSample class

This class is a specialized class derived from the *InputSample* class. It contains no data members, but contains a small set of member functions.

Member function	Description
Public	
WaveletInputSample(const SAMPLETYPE value=0, const unsigned index=0);	Default constructor that takes as input the value and index of the new sample.
WaveletInputSample(const WaveletInputSample &rhs);	Copy constructor that takes as input a reference to a WaveletInputSample.
virtual ∼WaveletInputSample();	Destructor.
virtual WaveletInputSample⟨SAMPLETYPE⟩ & operator=(const Sample⟨SAMPLETYPE⟩ &rhs);	Equal operator that takes as input a Sample and returns a reference to the WaveletInputSample.

A.4.5 WaveletOutputSample class

This class is a specialized class derived from the *OutputSample* class. It has data members and a accessor member functions for manipulating its data members.

Data member	Description
Protected	
int level;	Designates which level the sample belongs to in the decomposition.

Member function	Description
Public	
WaveletOutputSample();	Default constructor.
WaveletOutputSample(const WaveletOutputSample &rhs);	Copy constructor that takes as input a reference to a WaveletOutputSample.
WaveletOutputSample(const SAMPLETYPE value, const unsigned index);	Specialized constructor that takes as input the value and index. The level is not set and must be set later by another member function.
WaveletOutputSample(const SAMPLETYPE value, const int level, const unsigned index);	Specialized constructor that takes as input the value, the level at which it resides and the index number at that level.
virtual ~WaveletOutputSample();	Destructor.
virtual WaveletOutputSample & operator=(const Sample⟨SAMPLETYPE⟩ &rhs);	Equal operator that takes as input a reference to a Sample and returns a reference to the WaveletOutputSample.
WaveletOutputSample & operator=(const WaveletOutputSample &rhs);	Equal operator that takes as input a reference to a WaveletOutputSample and returns a reference to the WaveletOutputSample.
inline void SetSampleLevel(const int level);	Sets the level of the sample.
inline int GetSampleLevel() const;	Gets the level of the sample.
virtual ostream & Print(ostream &os) const;	Prints the contents of the class.
virtual ostream & operator≪(ostream &os) const;	Stream operator to print the contents of the class.

A.5 Sample block classes

There are multiple sample aggregating classes that serve different purposes. The purpose is highly dependent on the operation. In what follows, we describe the purpose of the class with a listing of its data members and member functions.

A.5.1 SampleBlock base class

The *SampleBlock* class serves as the base class for all sample blocks. The aggregating data structure is contained herein and most standard operations for manipulating the aggregated samples are functions of this class.

Data type	Description
Protected	
deque⟨SAMPLETYPE⟩ samples;	STL deque container type used for aggregating the samples. In some algorithms, the queue is accessed from the front and back.
unsigned blockindex;	The block index is used for maintaining order between subsequent arriving blocks.

Member function	Description
Public	
SampleBlock(const unsigned blockindex=0);	Default constructor with parameter blockindex.
SampleBlock(const SampleBlock &rhs);	Copy constructor.
SampleBlock(const deque⟨SAMPLETYPE⟩ &input);	Specialized constructor which takes as input an aggregated block of samples.
SampleBlock(const deque⟨SAMPLETYPE⟩ &input, const unsigned blockindex);	Specialized constructor which takes as input an aggregated block of samples and blockindex.
virtual ∼SampleBlock();	Destructor.
virtual SampleBlock & operator=(const SampleBlock &rhs);	Equal operator that takes as input a reference to a SampleBlock and returns a reference.
SampleBlock & operator+(const SampleBlock &rhs);	Addition operator adds the contents of the two blocks and returns a refernce to the result.
SampleBlock & operator+=(const SampleBlock &rhs);	Plus-equal operator adds the contents of the two blocks and returns a reference to the result.
inline SAMPLETYPE operator[](const unsigned i) const;	Operator provides random access to the queue indexed by i.

Member function	Description
Public	
inline void SetSamples(const deque⟨SAMPLETYPE⟩ &input);	Set the samples equal to the input sample deque.
virtual void SetSamples(const double* series, const int serlen);	Set the sample values equal to the input array with corresponding length.
inline void GetSamples(deque⟨SAMPLETYPE⟩ &buf) const;	Get the samples and store them into the input buffer and return passed by reference.
void GetSamples(double *series) const;	Get the samples and store them into the input array.
void GetSamples(deque⟨SAMPLETYPE⟩ &buf, const unsigned first, const unsigned last) const;	Get a range of samples based on the indices first and last and return by reference in the input argument buf.
inline SAMPLETYPE Front() const;	Obtain the sample from the front of the queue.
inline void PushSampleFront(const SAMPLETYPE &input);	Push new sample to the front of the queue.
inline void PopSampleFront();	Pop sample off the front of the queue.
inline SAMPLETYPE Back() const;	Obtain the sample from the back of the queue.
inline void PushSampleBack(const SAMPLETYPE &input);	Push new sample to the back of the queue.
inline void PopSampleBack();	Pop sample off the back of the queue.
inline void SetBlockIndex(const unsigned index);	Sets the block index.
inline unsigned GetBlockIndex() const;	Gets the block index.
void AppendBlockBack(const SampleBlock &block);	Append a block of samples to the back of the queue.
void AppendBlockFront(const SampleBlock &block);	Append a block of samples to the front of the queue.
void RemoveSamplesFront(const unsigned numsamples);	Remove a number of samples from the front of the queue.
void RemoveSamplesBack(const unsigned numsamples);	Remove a number of samples from the back of the queue.
inline void ClearBlock();	Clear the sample block.
inline bool Empty() const;	Checks if the block is empty and returns bool.
inline unsigned GetBlockSize() const;	Get the size of the sample block.
virtual SampleBlock* clone() const;	Clone the SampleBlock and return a pointer to it.
virtual ostream & Print(ostream &os) const;	Prints the contents of the class.
virtual ostream & operator≪(ostream &os) const;	Streaming operator for printing contents of class.

A.5.2 InputSampleBlock class

The *InputSampleBlock* class is essentially used to designate the block as type *input*. Input type classes will derive from this class which is derived from the *SampleBlock* class.

Member function	Description
Public	
InputSampleBlock();	Default constructor.
InputSampleBlock(const InputSampleBlock &rhs);	Copy constructor.
InputSampleBlock(const deque⟨SAMPLETYPE⟩ &input);	Specialized constructor that takes as input a reference to a deque of samples.
InputSampleBlock(const deque⟨SAMPLETYPE⟩ &input, const unsigned index);	Specialized constructor that takes as input a reference to a deque of samples and a block index.
virtual ∼InputSampleBlock();	Destructor.

A.5.3 OutputSampleBlock class

The *OutputSampleBlock* class is to designate the block as type *output*. Output type classes will derive from this class which is derived from the *SampleBlock* class.

Member function	Description
Public	
OutputSampleBlock();	Default constructor.
OutputSampleBlock(const OutputSampleBlock &rhs);	Copy constructor.
OutputSampleBlock(const deque⟨SAMPLETYPE⟩ &input);	Specialized constructor that takes as input a reference to a deque of samples.
OutputSampleBlock(const deque⟨SAMPLETYPE⟩ &input, const unsigned index);	Specialized constructor that takes as input a reference to a deque of samples and a block index.
virtual ∼OutputSampleBlock();	Destructor.

A.5.4 WaveletInputSampleBlock class

The *WaveletInputSampleBlock* class is derived from the *InputSampleBlock* class, and represents the input block type used for streaming block operations.

Member function	Description
Public	
WaveletInputSampleBlock();	Default constructor.
WaveletInputSampleBlock(const WaveletInputSampleBlock &rhs);	Copy constructor.
WaveletInputSampleBlock(const deque⟨SAMPLETYPE⟩ &input);	Specialized constructor that takes as input a reference to a deque of samples.
WaveletInputSampleBlock(const deque⟨SAMPLETYPE⟩ &input, const unsigned index);	Specialized constructor that takes as input a reference to a deque of samples and a block index.
virtual ∼WaveletInputSampleBlock();	Destructor.

A.5.5 WaveletOutputSampleBlock class

The *WaveletOutputSampleBlock* class is derived from the *OutputSampleBlock*, and represents that output block type used for streaming block operations. The samples contained within this block should all be tagged with the same level information.

Data member	Description
Protected	
int level;	The level information of the output sample block. The samples in the block should also be tagged with level information.

Member function	Description
Public	
WaveletOutputSampleBlock(const int level=0);	Default constructor that takes as input a default argument for the level.
WaveletOutputSampleBlock(const WaveletOutputSampleBlock &rhs);	Copy constructor.
WaveletOutputSampleBlock(const deque⟨SAMPLETYPE⟩ &input, const unsigned index);	Specialized constructor that takes as input a reference to a deque of samples and a block index.
virtual ∼WaveletOutputSampleBlock();	Destructor.
WaveletOutputSampleBlock & operator=(const WaveletOutputSampleBlock &rhs);	Equal operator that takes as input a reference to a *WaveletOutputSampleBlock*.
virtual WaveletOutputSampleBlock* clone() const;	Cloning operation that returns a pointer to the newly created object.
inline void SetBlockLevel(const int level);	Sets the level of the block.
inline int GetBlockLevel() const;	Gets the level of the block.
void SetSamples(const double* series, const int serlen);	Sets the sample values to the values in the input array with corresponding length.
inline void SetSamples(const deque⟨SAMPLETYPE⟩ &buf);	Sets samples to the contents of the input deque of samples.
bool AllSamplesLevelCorrect();	Returns true if all samples in the block are tagged with the appropriate level information.
void SetAllSamplesToCorrectLevel();	Sets all samples in the block to the correct level (that set by the data member *level*.

A.5.6 WaveletRandomOutputSampleBlock class

The class *WaveletRandomOutputSampleBlock* class assumes that the samples are randomly ordered and would need reordering to perform appropriate operations in the toolkit.

Member function	Description
Public	
WaveletRandomOutputSampleBlock();	Default constructor.
WaveletRandomOutputSampleBlock (const WaveletRandomOutputSampleBlock &rhs);	Copy constructor.
virtual ~WaveletRandomOutputSampleBlock();	Destructor.
virtual WaveletRandomOutputSampleBlock* clone() const;	Cloning function that returns a pointer to the newly created object.
inline void SetBlockLevelOfSample(const unsigned index, const int level);	Sets the level of the sample located by the input argument *index*.
inline int GetBlockLevelOfSample(const unsigned index) const;	Gets the level of the sample located by the input argument *index*.

A.5.7 DiscreteWaveletOutputSampleBlock class

The *DiscreteWaveletOutputSampleBlock* is an output block of samples for use with DWT operations. The encoding of this block is shown in Figure 29.

Member function	Description
Public	
DiscreteWaveletOutputSampleBlock (const unsigned numlevels=2, const int lowest_level=0, const TransformType tt=TRANSFORM);	Default constructor that takes as default parameters the number of levels, the lowest level represented in the block and the type of transform represented.
DiscreteWaveletOutputSampleBlock (const DiscreteWaveletOutputSampleBlock &rhs);	Copy constructor.
virtual ~DiscreteWaveletOutputSampleBlock();	Destructor.
virtual DiscreteWaveletOutputSampleBlock* clone() const;	Clone operation that returns a pointer to the newly constructed block.
inline int GetLowestLevel() const;	Returns the lowest output level represented in the block.
inline void SetLowestLevel(const int lowest_level);	Sets the lowest output level represented in the block.
inline unsigned GetNumberLevels() const;	Obtains the number of levels encoded in the block.
inline void SetNumberLevels(const unsigned numlevels);	Sets the number of levels encoded in the block.
inline TransformType GetTransformType() const;	Obtains the transform type encoded in the block.
inline void SetTransformType(const TransformType tt);	Sets the transform type.
void SetSamplesAtLevel(const deque⟨SAMPLETYPE⟩ &samps, const int level);	Set the samples from the input deque passed by reference at the appropriate level.
unsigned GetSamplesAtLevel(deque⟨SAMPLETYPE⟩ &out, const int level) const;	Gets the samples from a particular level and returns them by reference to the output deque.

A.6 Sampler classes

In this section, we provide the interfaces for performing up and down sample operations. The classes that are discussed in this section are the *DownSample* and *UpSample* class.

A.6.1 DownSample class

The *DownSample* class is used to resample a stream or block of samples to a new rate lower than the original. The rate must be an integer value.

Member Function	Description
Public	
DownSample(const unsigned rate=1);	Default constructor that takes as input the down sample rate.
DownSample(const DownSample &rhs);	Copy constructor.
virtual ∼DownSample();	Destructor.
DownSample & operator=(const DownSample &rhs);	Equal operator that takes as input a reference to a *DownSample* object.
inline void SetDownSampleRate(const unsigned rate);	Sets the down sample rate to *rate*.
inline unsigned GetDownSampleRate() const;	Gets the down sample rate.
inline void ResetState();	Resets the state of the down sampler object.
bool KeepSample();	Routine returns true of the sample should be kept and false if the sample can be thrown away.
void DownSampleBuffer(SampleBlock⟨SAMPLE⟩ &output, const SampleBlock⟨SAMPLE⟩ &input);	This routine downsamples a block of samples taken as input and returns the output sample block.
ostream & Print(ostream &os) const;	Prints the contents of the class.
ostream & operator≪(ostream &os) const;	Stream operator that prints the contents of the class.

A.6.2 UpSample class

The *UpSample* class is used to resample a stream or block of samples to a new rate greater than the original by adding zero samples in between samples of the original sampling rate. The rate must be an integer value.

Member Function	Description
Public	
UpSample(const unsigned rate=1);	The default constructor that takes as input the up sample rate.
UpSample(const UpSample &rhs);	Copy constructor.
virtual $\sim$UpSample();	Deconstructor.
UpSample & operator=(const UpSample &rhs);	Equal operator that takes as input a reference to an up sample object.
inline void SetUpSampleRate(const unsigned rate);	Sets the up sample rate to *rate*.
inline unsigned GetUpSampleRate() const;	Gets the up sample rate.
inline void ResetState();	Resets the state of the up sampler.
bool ZeroSample();	Routine returns true if the current sample time should be zero filled.
void UpSampleBuffer(SampleBlock$\langle$SAMPLE$\rangle$ &output, const SampleBlock$\langle$SAMPLE$\rangle$ &input);	This routine up samples a block of samples and returns the upsampled block as output.
ostream & Print(ostream &os) const;	Prints the contents of the class.
ostream & operator$\ll$(ostream &os) const;	Stream operator that prints the contents of the class.

A.7 Filter and coefficient classes

There are two types of classes that are discussed in this section. The *FIRFilter* class characterizes a finite-impulse response type filter. It uses the *CQFWaveletCoefficients* class to define the impulse response. Other filter types, and coefficients can be added to the system in the future.

A.7.1 FIRFilter class

The *FIRFilter* class contains operations and data structures to realize the functionality of an FIR filter that can be run in both sample and block modes.

Member function	Description
Public	
FIRFilter(const unsigned numcoefs=0);	Default constructor with default argument for number of coefficients.
FIRFilter(const FIRFilter &rhs);	Copy constructor.
FIRFilter(const unsigned numcoefs, const vector⟨double⟩ &coefs);	Specialized constructor that takes as arguments the number of coefficients and a reference to a vector of coefficient values.
virtual ∼FIRFilter();	Destructor.
FIRFilter & operator=(const FIRFilter &rhs);	Equal operator that takes as input a reference to an FIR filter.
void SetFilterCoefs(const vector⟨double⟩ &coefs);	Sets the filter coefficients to the values of the referenced vector.
void GetFilterCoefs(vector⟨double⟩ &coefs) const;	Gets the filter coefficients and passes them back by reference to the input vector.
void SetNumCoefs(const unsigned numcoefs);	Sets the number of coefficients.
inline unsigned GetNumCoefs() const;	Gets the number of coefficients.
void ClearDelayLine();	Clears the delay line.
void GetFilterOutput(Sample⟨SAMPLETYPE⟩ &out, const Sample⟨SAMPLETYPE⟩ &in);	Performs sample filter operations on the input sample and returns an output sample.
void GetFilterBufferOutput(SampleBlock⟨OUTSAMPLE⟩ &out, const SampleBlock⟨INSAMPLE⟩ &in);	Performs block filter operations on an input *SampleBlock* and return an output block of samples.
ostream & Print(ostream &os) const;	Prints the contents of the class.
ostream & operator≪(ostream &os) const;	Streaming operator that prints the contents of the class.

A.7.2 CQFWaveletCoefficients class

The *CQFWaveletCoefficients* class provides the filter coefficients for the low-pass and high-pass analysis and synthesis filters based on the CQF assumptions for perfect reconstruction. CQF filters are FIR filters, and therefore work directly with the *FIRFilter* class.

Member function	Description
Public	
CQFWaveletCoefficients(const WaveletType wt=DAUB2);	Default constructor that takes as input the wavelet type, defaulting to the Haar mother wavelet.
CQFWaveletCoefficients(const CQFWaveletCoefficients &rhs);	Copy constructor.
virtual ∼CQFWaveletCoefficients();	Deconstructor.
CQFWaveletCoefficients & operator=(const CQFWaveletCoefficients &rhs);	Equal operator that takes as input a reference to a *CQFWaveletCoefficients* object.
void Initialize(const WaveletType wt);	This function initializes the data members by calling the private function init().
void ChangeType(const WaveletType wt);	Changes the wavelet type and re-initializes the data members.
string GetWaveletName() const;	Returns a human readable string identifying the wavelet type.
unsigned GetNumCoefs() const;	Gets the number of coefficients.
void GetTransformCoefsLPF(vector⟨double⟩ & coefs) const;	Gets the analysis, low-pass filter coefficients, $g(-n) \Rightarrow G(z^{-1})$. The coefficients are returned in *coefs*.
void GetTransformCoefsHPF(vector⟨double⟩ & coefs) const;	Gets the analysis, high-pass filter coefficients, $h(-n) \Rightarrow H(z^{-1})$. The coefficients are returned in *coefs*.
void GetInverseCoefsLPF(vector⟨double⟩ & coefs) const;	Gets the synthesis, low-pass filter coefficients, $g(n) \Rightarrow G(z)$. The coefficients are returned in *coefs*.
void GetInverseCoefsHPF(vector⟨double⟩ & coefs) const;	Gets the synthesis, high-pass filter coefficients, $h(n) \Rightarrow H(z)$. The coefficients are returned in *coefs*.
ostream & Print(ostream &os) const;	Prints the contents of the class.
ostream & operator≪(ostream &os) const;	Stream operator for printing the contents of the class.

A.8 DelayBlock class

The primary function of the *Delay* class is to realize perfect reconstruction in the streaming transforms. It phase aligns the FIR filters with less coefficients to those with more coefficients. The class is simply a deque of samples that flow in one direction.

Member function	Description
Public	
DelayBlock(const unsigned numlevels=2, const int lowest_level=0, int* delay_vals=0);	Default constructor that takes as input the number of levels in the delay block, the lowest output level represented, and the delay value at each level.
DelayBlock(const DelayBlock &rhs);	Copy constructor.
virtual ∼DelayBlock();	Destructor.
DelayBlock & operator=(const DelayBlock &rhs);	Equal operator that takes as input a reference to a *DelayBlock*.
inline unsigned GetNumberLevels() const;	Gets the number of levels in in the delay block.
inline int GetLowestLevel() const;	Gets the lowest level represented in the delay block.
inline void SetLowestLevel(const int lowest_level);	Sets the lowest level represented.
inline unsigned GetDelayValueOfLevel(const int level) const;	Gets the delay value at a particular level.
bool SetDelayValueOfLevel(const int level, const unsigned delay);	Sets the delay value of a particular level.
bool ChangeDelayConfig(const unsigned numlevels, const int lowest_level, int* delay_vals);	Changes the delay configuration. It takes as inputs the number of levels, the lowest level and delay values.
bool ClearLevelDelayLine(const int level);	Clears the delay line at the specified level.
void ClearAllDelayLines();	Clears all delay lines.
bool StreamingSampleOperation(vector⟨SAMPLE⟩ &out, const vector⟨SAMPLE⟩ &in);	Performs the delay operation in sample streaming mode. The input and output are vectors indexed by the level.
unsigned StreamingBlockOperation (vector⟨WaveletOutputSampleBlock⟨SAMPLE⟩ ⟩ &outblock, const vector⟨WaveletOutputSampleBlock⟨SAMPLE⟩ ⟩ &inblock);	Performs the delay operation in block streaming mode. The input and output are vectors of sample blocks indexed by the level.
ostream & Print(ostream &os) const;	Prints the contents of the class
ostream & operator≪(ostream &os) const;	Stream operator for printing the contents of the class.

A.9 Jitter and jitter action classes

Jitter protection and recovery in communication systems is extremely important in order to reduce errors within the system. In Tsunami, we provide a single stream jitter protection class, *JitterProtectStream*, and a multiple stream jitter protection class, *JitterProtectMultiStream*. The multiple stream class uses the single stream class, and is used for recovering from jitter when the wavelet coefficients are streamed over the network. When jitter is detected and action must be taken, the toolkit provides two jitter action classes. These are the *ZeroFillAction* and the *InterpolateFillAction* class.

A.9.1 JitterProtectStream class

The *JitterProtectStream* class protects a stream of periodic samples from jitter and loss by first detecting the condition and then taking the appropriate action.

Member function	Description
Public	
JitterProtectStream (const unsigned backlog_thresh=DEFAULT_BACKLOG_THRESH);	Default constructor that takes a default argument for setting the backlog threshold.
JitterProtectStream(const JitterProtectStream &rhs);	Copy constructor.
virtual ∼JitterProtectStream();	Destructor.
JitterProtectStream & operator=(const JitterProtectStream &rhs);	Equal operator that takes as input a reference to a *JitterProtectStream*.
void ChangeBacklogThresh(const unsigned backlog_thresh);	Changes the backlog threshold on the fly.
inline unsigned GetBacklogThresh() const;	Gets the current backlog threshold.
inline void SetCurrentIndex(const unsigned curr_index);	Sets the current index to look for next.
inline unsigned GetCurrentIndex() const;	Gets the current index.
void JitterProtectSampleOperation(list⟨INSAMPLE⟩ &out, const INSAMPLE &in);	Sample operation jitter protection that takes as input a reference to a sample and returns a list of ordered output samples.
void JitterProtectBlockOperation(SampleBlock⟨INSAMPLE⟩ &out, const SampleBlock⟨INSAMPLE⟩ &in);	Block operation jitter protection that takes as input a reference to a sample block and returns a block of ordered samples.
ostream & Print(ostream &os) const;	Prints the contents of the class.
ostream & operator≪(ostream &os) const;	Stream operator used for printing the contents of the class.

A.9.2 JitterProtectMultiStream class

The *JitterProtectMultiStream* class uses the *JitterProtectStream* class to protect each of its multiple streams. Each stream has its own backlog threshold, and action is taken on each stream individually. This class is typically used to protect against jitter when the wavelet coefficients, the multi-level representation, is sent over a lossy network for reconstruction at an end system.

Member function	Description
Public	
JitterProtectMultiStreams(const unsigned numlevels=1, const int lowest_level=0, unsigned* backlogs=0);	Default constructor that takes as input the number of levels to be protected, the lowest level represented and backlogs for each level.
JitterProtectMultiStreams(const JitterProtectMultiStreams &rhs);	Copy constructor.
virtual ~JitterProtectMultiStreams();	Destructor.
JitterProtectMultiStreams & operator=(const JitterProtectMultiStreams &rhs);	Equal operator that takes as input a reference to a *JitterProtectMultiStreams*.
bool ChangeNumberOfLevels(const unsigned numlevels);	Changes the number of levels.
inline unsigned GetNumberOfLevels() const;	Gets the number of levels.
inline void ChangeLowestLevel(const int lowest_level);	Changes the value of the lowest level represented.
inline int GetLowestLevel() const;	Gets the lowest level.
void JitterProtectSampleOperation(vector⟨list⟨INSAMPLE⟩ ⟩ &out, const vector⟨INSAMPLE⟩ &in);	Sample operation multi-stream jitter protection that takes as input a vector of sampes indexed by level, and returns a vector of lists of ordered samples.
void JitterProtectBlockOperation (vector⟨WaveletOutputSampleBlock⟨INSAMPLE⟩ ⟩ &outblock, const vector⟨WaveletOutputSampleBlock⟨INSAMPLE⟩ ⟩ &inblock);	Block operation multi-stream jitter protection that takes as input a vector of sample blocks indexed by level and returns a vector of sample blocks.
ostream & Print(ostream &os) const;	Prints the contents of the class.
ostream & operator≪(ostream &os) const;	Stream operator used to print the contents of the class.

A.9.3 ZeroFillAction class

The *ZeroFillAction* class is used in conjunction with the jitter protection classes. When the jitter backlog threshold has been exceeded, a member function of this class, *JitterAction* is called to zero fill missing samples. Each of the action classes will have a member function called *JitterAction*.

Member Function	Description
Public	
static unsigned JitterAction(list⟨INSAMPLE⟩ &samples, const unsigned curr_index);	A function that fills missing samples by simply making them zero. It takes as input the current index, and a list of samples upon which to work on.

A.10 Stage classes

The stage classes described in this section create two-band filter banks that we have discussed earlier in the report. By chaining these two-band filter banks, a structure we call a *stage*, arbitrary types of decompositions are realized. The decompositions that we have created at the time of this writing are transform-type trees where the frequency has been sliced logarithmically in powers of two. In order to accomplish these structures, we use the *WaveletStageHelper*, which provides the commonality between the *ForwardWaveletStage* used for analysis and the *ReverseWaveletStage* used for synthesis.

A.10.1 WaveletStageHelper class

The *WaveletStageHelper* class is used to abstract out the common functionality between the forward and reverse stages. It uses an enumerated type to determine the stage direction, contains the wavelet type (i.e. D2, D4, ..., D20), the coefficients of the wavelet filter and the low-pass and high-pass filters associated with the stage type. In addition, it contains operations for performing the filtering in sample and block streaming modes.

Data type	Description
Protected	
StageType stagetype;	An enumerated type that designates the direction of the stage (FORWARD or REVERSE).
WaveletType wavetype;	The type of wavelet filter used. (i.e. DAUB2 the Haar).
CQFWaveletCoefficients wavecoefs;	The coefficients of the wavelet filter.
FIRFilter⟨SAMPLETYPE,OUTSAMPLE,INSAMPLE⟩ lowpass;	The low pass filter, either analyis or synthesis based on the stage type.
FIRFilter⟨SAMPLETYPE,OUTSAMPLE,INSAMPLE⟩ highpass;	The high pass filter, either analysis or synthesis based on the stage type.

Member Function	Description
Public	
WaveletStageHelper(const WaveletType wavetype=DAUB2, const StageType stagetype=FORWARD);	Default constructor that takes as input the wavelet type and the direction of the stage.
WaveletStageHelper(const WaveletStageHelper &rhs);	Copy constructor.
virtual ∼WaveletStageHelper();	Destructor.
WaveletStageHelper & operator=(const WaveletStageHelper &rhs);	Equal operator that takes as input a reference to a *WaveletStageHelper* and returns a reference to the newly created object.
void ChangeWaveletType(const WaveletType wavetype);	Changes the wavelet type.
string GetWaveletName() const;	Gets the human readable name of the wavelet type.
void SetFilterCoefsLPF(const vector⟨double⟩ &coefs);	Sets the filter coefficients for the analysis or synthesis low-pass filter.
unsigned GetNumCoefsLPF() const;	Gets the number of coefficients for the low-pass filter.
void PrintCoefsLPF() const;	Prints the coefficients of the low-pass filter.
void SetFilterCoefsHPF(const vector⟨double⟩ &coefs);	Sets the filter coefficients for the analysis or synthesis high-pass filter.
unsigned GetNumCoefsHPF() const;	Gets the number of coefficients for the high-pass filter.
void PrintCoefsHPF() const;	Prints the coefficients of the high-pass filter.
void ClearLPFDelayLine();	Clears the LPF delay line.
void LPFSampleOperation(Sample⟨SAMPLETYPE⟩ &out, const Sample⟨SAMPLETYPE⟩ &in);	Sample by sample filtering operation using the low-pass filter.
void LPFBufferOperation(SampleBlock⟨OUTSAMPLE⟩ &out, const SampleBlock⟨INSAMPLE⟩ &in);	Buffer filtering operation using the low-pass filter.
void ClearHPFDelayLine();	Clears the HPF delay line.
void HPFSampleOperation(Sample⟨SAMPLETYPE⟩ &out, const Sample⟨SAMPLETYPE⟩ &in);	Sample by sample filtering operation using the high-pass filter.
void HPFBufferOperation(SampleBlock⟨OUTSAMPLE⟩ &out, const SampleBlock⟩INSAMPLE⟨ &in);	Buffer filtering operation using the high-pass filter.
ostream & Print(ostream &os) const;	Print the contents of the class.
ostream & operator≪(ostream &os) const;	Stream operator that prints the contents of the class.

A.10.2 ForwardWaveletStage class

The *ForwardWaveletStage* class is a two-band stage that contains the *WaveletStageHelper* class for filter operations and two *DownSample* classes. It also contains data members for bookkeeping such as output level number to be tagged to output samples. The downsampler rates are configurable through this stage type. The stage has operations for running in sample or block streaming modes in order to decompose a time-domain signal into wavelet coefficients.

Data type	Description
Protected	
WaveletStageHelper⟨SAMPLETYPE, OUTSAMPLE, INSAMPLE⟩ stagehelp;	This provides the stage all filtering operations.
unsigned rate_l;	The down sample rate that precedes the low-pass filter.
unsigned rate_h;	The down sample rate that precedes the high-pass filter.
int outlevel_l;	The output level number of the low-pass branch.
int outlevel_h;	The output level number of the high-pass branch.
DownSample⟨OUTSAMPLE⟩ downsampler_l;	This provides the stage with the down sampler attached to the low-pass branch.
DownSample⟨OUTSAMPLE⟩ downsampler_h;	This provides the stage with the down sampler attached to the high-pass branch.

Member Function	Description
Public	
ForwardWaveletStage(const WaveletType wavetype=DAUB2);	Default contstructor that takes as input the wavelet type. It defaults to the Haar wavelet type.
ForwardWaveletStage(const ForwardWaveletStage &rhs);	Copy constructor.
ForwardWaveletStage(const WaveletType wavetype, const unsigned rate_l, const unsigned rate_h, const int outlevel_l, const int outlevel_h);	Specialized constructor that takes as input the wavelet type, the down sample rates of each filter branch and the level number of each branch.
virtual ∼ForwardWaveletStage();	Destructor.
ForwardWaveletStage & operator=(const ForwardWaveletStage &rhs);	Equal operator that takes as input a reference to a *ForwardWaveletStage* and returns a reference to this object.
ForwardWaveletStage* clone();	Clones the *ForwardWaveletStage*.
inline void SetDownSampleRateLow(const unsigned rate);	Sets the down sample rate of the low-pass branch.
inline unsigned GetDownSampleRateLow() const;	Gets the down sample rate of the low-pass branch.
inline void SetDownSampleRateHigh(const unsigned rate);	Sets the down sample rate of the high-pass branch.
inline unsigned GetDownSampleRateHigh() const;	Gets the down sample rate of the high-pass branch.
inline void SetOutputLevelLow(const int outlevel);	Sets the output level of the low-pass branch.
inline int GetOutputLevelLow() const;	Gets the output level of the low-pass branch.
inline void SetOutputLevelHigh(const int outlevel);	Sets the output level of the high-pass branch.
inline int GetOutputLevelHigh() const;	Gets the output level of the high-pass branch.
inline void ChangeWaveletType(const WaveletType wavetype);	Changes the filters of the stage to the type *wavetype*.
inline void ClearFilterDelayLines();	Clears the filter delay lines of both filters.
inline void ClearAllState();	Clears the filter delay lines and resets the state of the down samplers.
bool PerformSampleOperation (WaveletOutputSample⟨SAMPLETYPE⟩ &out_l, WaveletOutputSample⟨SAMPLETYPE⟩ &out_h, const Sample⟨SAMPLETYPE⟩ &in);	Sample operation which takes in a sample and returns output samples every $1/(rate_i \cdot f_s)$ sample times. It returns true if an output sample is ready.
unsigned PerformBlockOperation (WaveletOutputSampleBlock⟨OUTSAMPLE⟩ &out_l, WaveletOutputSampleBlock⟨OUTSAMPLE⟩ &out_h, const SampleBlock⟨INSAMPLE⟩ &in);	Block operation which takes in a sample block and returns two *WaveletOutputSampleBlock* and the output sample block length (both outputs same length).
ostream & Print(ostream &os) const;	Prints the contents of the class.
ostream & operator≪(ostream &os) const;	Stream operator that prints the contents of the class.

A.10.3 ReverseWaveletStage class

The *ReverseWaveletStage* class is a two-band stage that contains the *WaveletStageHelper* class for filter operations and two *UpSample* classes. The upsampler rates are configurable through this stage type. The stage has operations for running in sample or block streaming modes, and performs the reconstruction from the input wavelet coefficients.

Data type	Description
Protected	
WaveletStageHelper⟨SAMPLETYPE, OUTSAMPLE, INSAMPLE⟩ stagehelp;	This provides the stage all filtering operations.
unsigned rate_l;	The up sample rate that precedes the low-pass filter.
unsigned rate_h;	The up sample rate that precedes the high-pass filter.
UpSample⟨INSAMPLE⟩ upsampler_l;	This provides the stage with the up sampler attached to the low-pass branch.
UpSample⟨INSAMPLE⟩ upsampler_h;	This provides the stage with the up sampler attached to the high-pass branch.

Member Function	Description
Public	
ReverseWaveletStage (const WaveletType wavetype=DAUB2);	Default contstructor that takes as input the wavelet type. It defaults to the Haar wavelet type.
ReverseWaveletStage(const ReverseWaveletStage &rhs);	Copy constructor.
ReverseWaveletStage(const WaveletType wavetype, const unsigned rate_l, const unsigned rate_h);	Specialized constructor that takes as input the wavelet type and the up sample rates of each filter branch.
virtual ∼ReverseWaveletStage();	Destructor.
ReverseWaveletStage & operator=(const ReverseWaveletStage &rhs);	Equal operator that takes as input a reference to a *ReverseWaveletStage* and returns a reference to this object.
ReverseWaveletStage* clone();	Clones the *ReverseWaveletStage*.
inline void SetUpSampleRateLow(const unsigned rate);	Sets the up sample rate of the low-pass branch.
inline unsigned GetUpSampleRateLow() const;	Gets the up sample rate of the low-pass branch.
inline void SetUpSampleRateHigh(const unsigned rate);	Sets the up sample rate of the high-pass branch.
inline unsigned GetUpSampleRateHigh() const;	Gets the up sample rate of the high-pass branch.
inline void ChangeWaveletType (const WaveletType wavetype);	Changes the filters of the stage to the type *wavetype*.
inline void ClearFilterDelayLines();	Clears the filter delay lines of both filters.
inline void ClearAllState();	Clears the filter delay lines and resets the state of the up samplers.
bool PerformSampleOperation (vector⟨OUTSAMPLE⟩ &out, const Sample⟨SAMPLETYPE⟩ &in_l, const Sample⟨SAMPLETYPE⟩ &in_h);	Sample operation that takes in a sample on each branch and produces twice as many output samples. It returns true if there are samples ready.
unsigned PerformBlockOperation (SampleBlock⟨OUTSAMPLE⟩ &out, const SampleBlock⟨INSAMPLE⟩ &in_l, const SampleBlock⟨INSAMPLE⟩ &in_h);	Block operation that takes as input a sample block on each branch and returns an output sample block twice as long as the input sample blocks and the output sample block length.
ostream & Print(ostream &os) const;	Prints the contents of the class.
ostream & operator≪(ostream &os) const;	Stream operator that prints the contents of the class.

A.11 Transform classes

In this section, we discuss the transforms that are provided in the Tsunami toolkit. These include the statically structured transform classes, *StaticForwardWaveletTransform* and *StaticReverseWaveletTransform*, the dynamically structured transform classes, *DynamicForwardWaveletTransform* and *DynamicReverseWaveletTransform* subclassed from the static transforms, and the discrete transform classes, *ForwardDiscreteWaveletTransform* and *ReverseDiscreteWaveletTransform*.

A.11.1 StaticForwardWaveletTransform class

The *StaticForwardWaveletTransform* class provides a statically structured, streaming wavelet transform. It includes a number of *ForwardWaveletStages* designated by the data member *numstages*, arrays for indexing the output detail and approximation samples, and a notion of the lowest output level in order to keep track of output level numbering.

Data type	Description
Protected	
unsigned numstages;	The number of stages to include in the transform.
unsigned numlevels;	The number of level decomposition. $numlevels = numstages + 1$
int lowest_outlvl;	The lowest output level number in the decomposition. Also the output level number of the highest frequency band.
unsigned index_a[MAX_STAGES+1];	An array of indices for approximation samples indexed by the level number offset.
unsigned index_d[MAX_STAGES+1];	An array of indices for detail samples indexed by the level number offset.
ForwardWaveletStage⟨SAMPLETYPE, OUTSAMPLE, INSAMPLE⟩* first_stage;	The first stage of the decomposition that is parameterized by *INSAMPLE* as the input type and *OUTSAMPLE* as the output type.
vector⟨ForwardWaveletStage⟨SAMPLETYPE, OUTSAMPLE, OUTSAMPLE⟩ *⟩ stages;	The remaining stages of the decompositon that are parameterized by *OUTSAMPLE* as the input type and *OUTSAMPLE* as the output type.

Member Function	Description
Public	
StaticForwardWaveletTransform (const unsigned numstages=1, const int lowest_outlvl=0);	Default constructor that takes as input the number of stages in the decomposition and the lowest output level.
StaticForwardWaveletTransform (const StaticForwardWaveletTransform &rhs);	Copy constructor.
StaticForwardWaveletTransform(const unsigned numstages, const WaveletType wavetype, const unsigned rate_l, const unsigned rate_h, const int lowest_outlvl);	Specialized constructor that takes as input the number of stages, the wavelet type, the down sample rates of each branch and the lowest output level.
virtual ∼StaticForwardWaveletTransform();	Destructor.
StaticForwardWaveletTransform & operator=(const StaticForwardWaveletTransform &rhs);	Equal operator that takes as input a reference to a *StaticForwardWaveletTransform*.
inline unsigned GetNumberStages() const;	Gets the number of stages.
bool ChangeNumberStages(const unsigned numstages);	Changes the number of stages and also clears all state.
bool ChangeNumberStages(const unsigned numstages, const WaveletType wavetype, const unsigned rate_l, const unsigned rate_h, const int lowest_outlvl);	Changes the number of stages and sets the new wavelet type, the down sample rates and the lowest output level.
inline int GetLowestOutputLevel() const;	Gets the lowest output level.
inline void SetLowestOutputLevel(const int lowest_outlvl);	Sets the number of the lowest output level.
inline unsigned GetIndexNumberOfApproxLevel (const int level) const;	Gets the index number of the current approximation sample at *level*.
inline unsigned GetIndexNumberOfDetailLevel (const int level) const;	Gets the index number of the current detail sample at *level*.
inline void SetIndexNumberOfApproxLevel (const int level, const unsigned newindex);	Sets the index number of the current approximation sample to *newindex* at *level*.
inline void SetIndexNumberOfDetailLevel (const int level, const unsigned newindex);	Sets the index number of the current detail sample to *newindex* at *level*.
ostream & Print(ostream &os) const;	Prints the contents of the class.
ostream & operator≪(ostream &os) const;	Stream operator that prints the contents of the class.

Member Function	Description
Public	
bool StreamingSampleOperation(vector⟨OUTSAMPLE⟩ &approx_out, vector⟨OUTSAMPLE⟩ &detail_out, const Sample⟨SAMPLETYPE⟩ &in);	Streaming sample operation that provides all approximations and detail signals.
bool StreamingTransformSampleOperation(vector⟨OUTSAMPLE⟩ &out, const Sample⟨SAMPLETYPE⟩ &in);	Streaming sample operation that provides one approximation and rest detail signals.
bool StreamingApproxSampleOperation(vector⟨OUTSAMPLE⟩ &approx_out, const Sample⟨SAMPLETYPE⟩ &in);	Streaming sample operation that provides only the approximations.
bool StreamingDetailSampleOperation(vector⟨OUTSAMPLE⟩ &detail_out, const Sample⟨SAMPLETYPE⟩ &in);	Streaming sample operation that provides only the details.
bool StreamingMixedSampleOperation(vector⟨OUTSAMPLE⟩ &approx_out, vector⟨OUTSAMPLE⟩ &detail_out, const Sample⟨SAMPLETYPE⟩ &in, const SignalSpec &spec);	Streaming sample operation that provides a mix of details and approximations based on the signal spec.
unsigned StreamingBlockOperation (vector⟨WaveletOutputSampleBlock⟨OUTSAMPLE⟩ ⟩ &approx_outblock, vector⟨WaveletOutputSampleBlock⟨OUTSAMPLE⟩ ⟩ &detail_outblock, const SampleBlock⟨INSAMPLE⟩ &inblock);	Streaming block operation that provides all approximations and detail signals.
unsigned StreamingTransformBlockOperation (vector⟨WaveletOutputSampleBlock⟨OUTSAMPLE⟩ ⟩ &outblock, const SampleBlock⟨INSAMPLE⟩ &inblock);	Streaming block operation that provides one approximation and rest detail signals.
unsigned StreamingApproxBlockOperation (vector⟨WaveletOutputSampleBlock⟨OUTSAMPLE⟩ ⟩ &approx_outblock, const SampleBlock⟨INSAMPLE⟩ &inblock);	Streaming block operation that provides only the approximations.
unsigned StreamingDetailBlockOperation (vector⟨WaveletOutputSampleBlock⟨OUTSAMPLE⟩ ⟩ &detail_outblock, const SampleBlock⟨INSAMPLE⟩ &inblock);	Streaming block operation that provides only the detail signals.
unsigned StreamingMixedBlockOperation (vector⟨WaveletOutputSampleBlock⟨OUTSAMPLE⟩ ⟩ &approx_outblock, vector⟨WaveletOutputSampleBlock⟨OUTSAMPLE⟩ ⟩ &detail_outblock, const SampleBlock⟨INSAMPLE⟩ &inblock, const SignalSpec &spec);	Streaming block operation that provides a mix of details and approximations based on the signal spec.

A.11.2 StaticReverseWaveletTransform class

The *StaticReverseWaveletTransform* is the statically structured dual to the *StaticForwardWaveletTransform*, and is used for reconstruction from the wavelet coefficients output from the forward transform. This class consists of data members for keeping track of the number of stages, the number of levels, outgoing indices, incoming indices, buffers for the input signals that arrive at varying stream rates, buffers in between stages for dealing with different stream rates, and a number of *ReverseWaveletStages* to perform the filtering and upsampling at each stage. Like the forward transform, this class can be run in sample or block streaming modes of operation.

Data type	Description
Protected	
unsigned numstages;	The number of stages to include in the reconstruction.
unsigned numlevels;	The number of level reconstruction. $numlevels = numstages + 1$
int lowest_inlvl;	The lowest input level represented by the incoming signals.
unsigned index;	Outgoing indice counter to index the output samples.
unsigned indices[MAX_STAGES+1];	An array of input indices for determing when enough samples have arrived so that the reconstruction can be properly performed.
unsigned sampletime;	Keeps track of the sampletime based on incoming samples in order to zero-fill missing samples.
bool sync;	True if the incoming indices have been properly synchronized. False otherwise.
unsigned sync_level;	The level upon which to synchronize the sampletime calculations.
vector⟨SampleBlock⟨INSAMPLE⟩ *⟩ insignals;	Input buffers for the incoming input signals. This is required because each of the streams arrive at varying rates.
vector⟨SampleBlock⟨INSAMPLE⟩ *⟩ intersignals;	Buffers that sit in between subsequent stages. This is required because signals between stages arrive at varying rates.
vector⟨ReverseWaveletStage⟨SAMPLETYPE, INSAMPLE, INSAMPLE⟩ *⟩ stages;	A vector of *ReverseWaveletStages* indexed by the level number. The stages are parameterized by input type *INSAMPLE* and output type *INSAMPLE*.
ReverseWaveletStage⟨SAMPLETYPE, OUTSAMPLE, INSAMPLE⟩* last_stage;	The last *ReverseWaveletStage* that is parameterized by input type *INSAMPLE* and output type *OUTSAMPLE*. This stage converts the input from *INSAMPLE* to the output type *OUTSAMPLE*.

Member Function	Description
Protected	
inline void ClearAllDelayLines();	Clears all filter delay line.
inline bool SamplePairReady (const SampleBlock⟨INSAMPLE⟩ &block_l, const SampleBlock⟨INSAMPLE⟩ &block_h) const;	Returns true of there are samples in each block that are ready to be run through a stage.
inline bool BlockPairReady (const SampleBlock⟨INSAMPLE⟩ &block_l, const SampleBlock⟨INSAMPLE⟩ &block_h) const;	Returns true if the blocks have samples that are ready to be run through a stage.
void AddRemainingBlockToInsignals (const SampleBlock⟨INSAMPLE⟩ &block, const unsigned minsize, const unsigned level);	Adds the remaining samples from a sample block that have not been run through the stage into the insignals buffer at index *level*.
void AddBlockToInsignals (const SampleBlock⟨INSAMPLE⟩ &block, const unsigned level);	Adds the samples from the sample block into the insignal buffer at a particular level.
void AddRemainingBlockToIntersignals (const SampleBlock⟨INSAMPLE⟩ &block, const unsigned minsize, const unsigned level);	Adds the remaining samples from a sample block that have not been run through a stage into the intersignals buffer at index *level*.
void AddBlockToIntersignals (const SampleBlock⟨INSAMPLE⟩ &block, const unsigned level);	Adds the samples from the sample block into the intersignal buffer at a particular level.
void AddZeroSamplesToInput (vector⟨INSAMPLE⟩ &zeros, const vector⟨int⟩ &zerolevels);	Adds zero samples to the input sample data structures.
Public	
StaticReverseWaveletTransform (const unsigned numstages=1, const int lowest_inlvl=0);	Default constructor that takes as input the number of stages in the reconstruction and the lowest input level.
StaticReverseWaveletTransform (const StaticReverseWaveletTransform &rhs);	Copy constructor.
StaticReverseWaveletTransform (const unsigned numstages, const WaveletType wavetype, const unsigned rate_l, const unsigned rate_h, const int lowest_inlvl);	Specialized constructor that takes as input the numbber of stages, the wavelet type, the up sample rates of each branch and the lowest input level.
virtual ∼StaticReverseWaveletTransform();	Destructor.
StaticReverseWaveletTransform & operator=(const StaticReverseWaveletTransform &rhs);	Equal operator that takes as input a reference to a *StaticReverseWaveletTransform*.

Member Function	Description
Public	
inline unsigned GetNumberStages() const;	Gets the number of stages.
bool ChangeNumberStages(const unsigned numstages);	Changes the number of stages to current type and clears all state.
bool ChangeNumberStages(const unsigned numstages, const WaveletType wavetype, const unsigned rate_l, const unsigned rate_h, const int lowest_inlvl);	Changes the number of stages and sets the new wavelet type, the up sample rates and the lowest input level.
inline int GetLowestInputLevel() const;	Gets the lowest input level.
inline void SetLowestInputLevel(const int lowest_inlvl);	Sets the number of the lowest input level.
inline unsigned GetIndexNumber() const;	Gets the current output index number.
inline void SetIndexNumber(const unsigned index);	Sets the current output index number.
inline void ClearIncomingIndices();	Clear the incoming indices.
inline unsigned GetSampleTime() const;	Obtain the sampletime estimate.
inline void SetSampleTime(const unsigned sampletime);	Set the sampletime estimate.
inline bool GetSyncStatus() const;	Obtain the synchronization status.
inline void SetSyncStatus(const bool sync);	Set the synchronization staus.
bool StreamingTransformSampleOperation (vector⟨OUTSAMPLE⟩ &out, const vector⟨INSAMPLE⟩ &in);	Streaming sample operation that reconstructs using one approximation signal and the rest detail signals.
bool StreamingTransformZeroFillSampleOperation (vector⟨OUTSAMPLE⟩ &out, const vector⟨INSAMPLE⟩ &in, const vector⟨int⟩ &zerolevels);	Streaming sample operation that reconstructs using one approximation signal and the rest detail signals with zero filling levels designated by the *zerolevels* spec.
bool StreamingMixedSampleOperation (vector⟨OUTSAMPLE⟩ &out, const vector⟨INSAMPLE⟨ &approx_in, const vector⟨INSAMPLE⟩ &detail_in, const SignalSpec &spec);	Streaming sample operation that reconstructs using a mix of approximations and details based on the signal spec.

Member Function	Description
Public	
unsigned StreamingTransformBlockOperation (SampleBlock⟨OUTSAMPLE⟩ &outblock, const vector⟨WaveletOutputSampleBlock⟨INSAMPLE⟩ ⟩ &inblock);	Streaming block operation that reconstructs using one approximation signal and the rest detail signals.
unsigned StreamingTransformZeroFillBlockOperation (SampleBlock⟨OUTSAMPLE⟩ &outblock, const vector⟨WaveletOutputSampleBlock⟨INSAMPLE⟩ ⟩ &inblock, const vector⟨int⟩ &zerolevels);	Streaming block operation that reconstructs using one approximation signal and the rest detail signals with zero filling levels designated by the *zerolevels* spec.
unsigned StreamingMixedBlockOperation (SampleBlock⟨OUTSAMPLE⟩ &outblock, const vector⟨WaveletOutputSampleBlock⟨INSAMPLE⟩ ⟩ &approx_block, const vector⟨WaveletOutputSampleBlock⟨INSAMPLE⟩ ⟩ &detail_block, const SignalSpec &spec);	Streaming block operation that reconstructs using a mix of approximations and details based on the signal spec.
ostream & Print(ostream &os) const;	Prints the contents of the class.
ostream & operator≪(ostream &os) const;	Stream operator that prints the contents of the class.

A.11.3 DynamicForwardWaveletTransform class

The *DynamicForwardWaveletTransform* class is subclassed from the *StaticForwardWaveletTransform* class, but provides the user with dynamic operations that add stages or remove stages without clearing the state of the class. This allows the transform to be shaped at run-time to the signature of the input signal being transformed.

Member Function	Description
Public	
DynamicForwardWaveletTransform();	Default constructor.
DynamicForwardWaveletTransform (const DynamicForwardWaveletTransform &rhs);	Copy constructor.
DynamicForwardWaveletTransform (const unsigned numstages=1, const int lowest_outlvl=0);	Specialized constructor that takes as input the number of stages and the lowest output level.
DynamicForwardWaveletTransform (const unsigned numstages, const WaveletType wavetype, const unsigned rate_l, const unsigned rate_h, const int lowest_outlvl);	Specialized constructor that takes as input the number of stages, wavelet type for each stage, the downsample rates, and the lowest output level.
virtual ∼DynamicForwardWaveletTransform();	Destructor.
bool AddStage();	Dynamically adds a stage of the existing type to the structure. This can be done at run-time, and all existing state remains the same.
bool AddStage(const WaveletType wavetype, const unsigned rate_l, const unsigned rate_h);	Same as above except that the stage that is added is specified by the wavelet type and down sample rates.
bool RemoveStage();	Removes a stage from the top, lowest frequency band of the structure.
bool ChangeAllWaveletTypes (const WaveletType wavetype);	Change all wavelet types in all stages.
bool ChangeStageWaveletTypes (const WaveletType wavetype, const unsigned stagenum);	Changes the wavelet type of a particular stage.
bool ChangeStructure (const unsigned new_numstages, const WaveletType new_wavetype);	Changes the structure to a new number of stages and a new wavelet type.
ostream & operator≪(ostream &os) const;	Stream operator that prints the contents of the class.

A.11.4 DynamicReverseWaveletTransform class

The *DynamicReverseWaveletTransform* class is subclassed from the *StaticReverseWaveletTransform* class, but provides the user with dynamic operations that add stages or remove stages without clearing the state of the class. This allows the reconstruction to be shaped at run-time to the signature of the wavelet coefficients that are streaming into the structure. As an example, if a set of levels are producing little to no energy in the wavelet coefficients, these stages might be dynamically removed at run-time.

Member Function	Description
Public	
DynamicReverseWaveletTransform();	Default constructor.
DynamicReverseWaveletTransform (const DynamicReverseWaveletTransform &rhs);	Copy constructor.
DynamicReverseWaveletTransform (const unsigned numstages=1, const int lowest_inlvl=0);	Specialized constructor that takes as input the number of stages and the lowest input level.
DynamicReverseWaveletTransform (const unsigned numstages, const WaveletType wavetype, const unsigned rate_l, const unsigned rate_h, const int lowest_inlvl);	Specialized constructor that takes as input the number of stages, wavelet type for each stage, the upsample rates, and the lowest input level.
virtual ∼DynamicReverseWaveletTransform() ;	Destructor.
bool AddStage();	Dynamically adds a stage of the existing type to the structure. This can be done at run-time, and all existing state remains the same.
bool AddStage(const WaveletType wavetype, const unsigned rate_l, const unsigned rate_h);	Same as above except that the stage that is added is specified by the wavelet type and up sample rates.
bool RemoveStage();	Removes a stage from the top, lowest frequency band of the structure.
bool ChangeAllWaveletTypes (const WaveletType wavetype);	Change all wavelet types in all stages.
bool ChangeStageWaveletTypes (const WaveletType wavetype, const unsigned stagenum);	Changes the wavelet type of a particular stage.
bool ChangeStructure (const unsigned new_numstages, const WaveletType new_wavetype);	Changes the structure to a new number of stages and a new wavelet type.
ostream & operator≪(ostream &os) const;	Stream operator that prints the contents of the class.

A.11.5 ForwardDiscreteWaveletTransform class

The *ForwardDiscreteWaveletTransform* class implements the Discrete Wavelet Transform (DWT). The operations in this class are run in block mode only, and the number of levels are a function of the input block size.

Member Function	Description
Public	
ForwardDiscreteWaveletTransform (const WaveletType wavetype=DAUB2, const int lowest_outlvl=0);	Default constructor that takes as input the wavelet type and the number of the lowest output level.
ForwardDiscreteWaveletTransform (const ForwardDiscreteWaveletTransform &rhs);	Copy constructor.
virtual ~ForwardDiscreteWaveletTransform();	Destructor.
ForwardDiscreteWaveletTransform & operator=(const ForwardDiscreteWaveletTransform &rhs);	Equal operator that takes as input a reference to a *ForwardDiscreteWaveletTransform*.
inline int GetLowestOutputLevel() const;	Gets the number of the lowest output level.
inline void SetLowestOutputLevel(const int lowest_outlvl);	Sets the number of the lowest output level.
inline unsigned GetIndexNumberOfApproxLevel (const int level) const;	Gets sample index number of approximation level designated by *level*.
inline unsigned GetIndexNumberOfDetailLevel (const int level) const;	Gets sample index number of detail level designated by *level*.
inline void SetIndexNumberOfApproxLevel (const int level, const unsigned newindex);	Sets the next sample index number of approximation level designated by *level*.
inline void SetIndexNumberOfDetailLevel (const int level, const unsigned newindex);	Sets the next sample index number of detail level designated by *level*.
inline WaveletType GetWaveletType() const;	Gets the wavelet type used in the DWT.
bool ChangeWaveletType(const WaveletType wavetype);	Sets the wavelet type used in the DWT.

Member Function	Description
Public	
unsigned DiscreteWaveletOperation (DiscreteWaveletOutputSampleBlock⟨OUTSAMPLE⟩ &approxblock, DiscreteWaveletOutputSampleBlock⟨OUTSAMPLE⟩ &detailblock, const SampleBlock⟨INSAMPLE⟩ &inblock);	Discrete wavelet operation that provides all of the approximation and detail signals.
unsigned DiscreteWaveletTransformOperation (DiscreteWaveletOutputSampleBlock⟨OUTSAMPLE⟩ &outblock, const SampleBlock⟨INSAMPLE⟩ &inblock);	Discrete wavelet transform operation that provides one approximation and the rest details.
unsigned DiscreteWaveletApproxOperation (DiscreteWaveletOutputSampleBlock⟨OUTSAMPLE⟩ &approxblock, const SampleBlock⟨INSAMPLE⟩ &inblock);	Discrete wavelet operation that provides approximation signals only.
unsigned DiscreteWaveletDetailOperation (DiscreteWaveletOutputSampleBlock⟨OUTSAMPLE⟩ &detailblock, const SampleBlock⟨INSAMPLE⟩ &inblock);	Discrete wavelet operation that provides detail signals only.
unsigned DiscreteWaveletMixedOperation (vector⟨WaveletOutputSampleBlock⟨OUTSAMPLE⟩ ⟩ &approxblock, vector⟨WaveletOutputSampleBlock⟨OUTSAMPLE⟩ ⟩ &detailblock, const SampleBlock⟨INSAMPLE⟩ &inblock, const SignalSpec &spec);	Discrete wavelet operation that provides a mix of approximation and detail signals based on the input signal specification.
ostream & operator≪(ostream &os) const;	Stream operator that prints the contents of the class.

A.11.6 ReverseDiscreteWaveletTransform class

The *ReverseDiscreteWaveletTransform* class implements the Inverse Discrete Wavelet Transform (IDWT). The operations in this class are run in block mode only, and the number of levels are a function of the input block size. It typically takes an encoded block of samples upon which it works to create the reconstructed time-domain signal.

Member Function	Description
Public	
ReverseDiscreteWaveletTransform (const WaveletType wavetype=DAUB2); const int lowest_inlvl=0);	Default constructor that takes as input the wavelet type and lowest input level.
ReverseDiscreteWaveletTransform (const ReverseDiscreteWaveletTransform &rhs);	Copy constructor.
virtual ReverseDiscreteWaveletTransform();	Destructor.
ReverseDiscreteWaveletTransform & operator=(const ReverseDiscreteWaveletTransform &rhs);	Equal operator that takes as input a reference to a *ReverseDiscrete-WaveletTransform.*
inline unsigned GetIndexNumber() const;	Gets the current sample index number.
inline void SetIndexNumber(const unsigned newindex);	Sets the current next output sample index number.
inline int GetLowestInputLevel() const;	Obtains the lowest input level.
inline void SetLowestInputLevel(const int lowest_inlvl);	Sets the lowest input level.
inline WaveletType GetWaveletType() const;	Get the current wavelet type.
bool ChangeWaveletType(const WaveletType wavetype);	Change the current wavelet type.
bool DiscreteWaveletTransformOperation (SampleBlock⟨OUTSAMPLE⟩ &outblock, const DiscreteWaveletOutputSampleBlock⟨INSAMPLE⟩ &inblock);	Inverse discrete wavelet transform operation that reconstructs the time-domain signal from an input sample block.
bool DiscreteWaveletTransformZeroFillOperation (SampleBlock⟨OUTSAMPLE⟩ &outblock, const DiscreteWaveletOutputSampleBlock⟨INSAMPLE⟩ &inblock); const vector⟨int⟩ &zerolevels);	Inverse discrete wavelet transform operation that reconstructs the time-domain signal from an input sample block with zero filling based on the zero fill specification.
bool DiscreteWaveletMixedOperation (SampleBlock⟨OUTSAMPLE⟩ &outblock, const vector⟨WaveletOutputSampleBlock⟨INSAMPLE⟩ ⟩ &approxblock, const vector⟨WaveletOutputSampleBlock⟨INSAMPLE⟩ ⟩ &detailblock, const unsigned numlevels);	Inverse discrete wavelet transform operation that reconstructs the time-domain signal from a mix of approximations and details.
ostream & operator≪(ostream &os) const;	Stream operator that prints the contents of the class.

References

[1] P. Abry, P. Flandrin, M. S. Taqqu, and D. Veitch. *Long-Range Dependence: Theory and Applications*, chapter Self-similarity and long range dependence through the wavelet lens. Birkhauser, 2002.

[2] P. Abry, D. Veitch, and P. Flandrin. Long-range dependence: Revisiting aggregation with wavelets. *Journal of Time Series Analysis*, 19(3):253–266, May 1998.

[3] G. Booch. *Object-oriented analysis and design*. Addison Wesley Longman, Inc., 2nd edition, 1994.

[4] I. Daubechies. *Ten Lectures on Wavelets*. Society for Industrial and Applied Mathematics (SIAM), 1999.

[5] P. A. Dinda and D. R. O'Hallaron. An extensible toolkit for resource prediction in distributed systems. Technical Report CMU-CS-99-138, School of Computer Science, Carnegie Mellon University, July 1999.

[6] A. Feldman, A. C. Gilbert, and W. Willinger. Data networks as cascades: Investigating the multifractal nature of internet WAN traffic. In *Proceedings of ACM SIGCOMM '98*, pages 25–38, 1998.

[7] A. Feldmann, A. Gilbert, P. Huang, and W. Willinger. Dynamics of ip traffic: a study of the role of variability and the impact of control. In *Proceedings of the ACM SIGCOMM 1999*, CAmbridge, MA, August 29 - September 1 1999.

[8] P. Flandrin. Wavelet analysis and synthesis of fractional brownian motion. *IEEE Transactions on Information Theory*, 38:910–916, March 1992.

[9] P. Huang, A. Feldmann, and W. Willinger. A non-intrusive, wavelet based approach to detecting network performance problems. In *Proceeding of ACM SIGCOMM Internet Measurement Workshop 2001*, San Francisco, CA, November 2001.

[10] M. W. Knop, P. K. Paritosh, P. A. Dinda, and J. M. Schopf. Windows performance monitoring and data reduction using watchtower and argus. Technical Report NWU-CS-01-6, Department of Computer Science, Northwestern University, June 2001.

[11] B. Lowekamp, N. Miller, D. Sutherland, T. Gross, P. Steenkiste, and J. Subhlok. A resource monitoring system for network-aware applications. In *Proceedings of the 7th IEEE International Symposium on High Performance Distributed Computing (HPDC)*, pages 189–196. IEEE, July 1998.

[12] S. Mallat. Multiresolution approximation and wavelets. *Transactions American Mathematics Society*, pages 69–88, 1989.

[13] M. Misiti, Y. Misiti, G. Oppenheim, and J. Poggi. *Wavelet Toolbox User's Guide*. The Mathworks, Inc., 3 Apple Hill Drive, Natick, MA 01760-2098, version 2.2 edition, July 2002.

[14] K. Nayebi, T. P. Barnwell, and M. J. T. Smith. Low delay fir filter banks: Design and evaluation. *IEEE Transactions on Signal Processing*, 42(1):24–31, January 1994.

[15] D. E. Newland. *An Introduction to Random Vibrations, Spectral and Wavelet Analysis*. Addison Wesley Longman Limited, 1993.

[16] Y. Qiao, J. Skicewicz, and P. Dinda. Multiscale predictability of network traffic. Technical Report TR NWU-CS-02-13, Northwestern University, Evanston, IL, 2003.

[17] R. Riedi, M. Crouse, V. Ribeiro, and R. Baraniuk. A multifractal wavelet model with application to network traffic. *IEEE Transactions on Information Theory*, 45(3):992–1019, April 1999.

[18] M. Roughan, D. Veitch, and P. Abry. On-line estimation of the parameters of long-range dependence. In *Proceedings Globecom 1998*, volume 6, pages 3716–3721, November 1998.

[19] G. Schuller. Time varying filter banks with variable system delay. In *IEEE International Conference on Acoustics, Speech, and Signal Proecessing (ICASSP)*, Munich, Germany, April 21-24 1997.

[20] G. D. T. Schuller and T. Karp. Modulated filter banks with arbitrary system delay: Efficient implementations and the time-varying case. *IEEE Transactions on Signal Processing*, 48(3):737–748, March 2000.

[21] J. Skicewicz, P. Dinda, and J. Schopf. Multi-resolution resource behavior queries using wavelets. In *Proceedings of the 10th Intl. Symp. on High Performance Distriubuted Computing (HPDC-10)*, pages 395–405, San Francisco, CA, August 2001.

[22] M. J. T. Smith and T. P. Barnwell. Exact reconstruction techniques for tree structured subband coders. *IEEE Transactions on Acoustics, Speech and Signal Processing*, ASSP-34(1):434–441, June 1986.

[23] I. Sodagar, K. Nayebi, and T. P. Barnwell. Time-varying filter banks and wavelets. *IEEE Transactions on Signal Processing*, 42(11):2983–2996, November 1994.

[24] I. The Mathworks. The matlab and simulink web site. http://www.mathworks.com.

[25] P. P. Vaidyanathan and I. Djokovic. *The Circuits and Filters Handbook*, chapter 6, Wavelet Transforms. CRC Press and IEEE Press, 1995.

[26] R. Wolski. Forecasting network performance to support dynamic scheduling using the network weather service. In *Proceedings of the 6th High-Performance Distributed Computing Conference (HPDC97)*, pages 316–325, August 1997. extended version available as UCSD Technical Report TR-CS96-494.