# Tiresias: A comprehensive framework for describing, analyzing, and visualizing connectivity data over time

Ryan Noon
Department of Computer Science
Stanford University
rmnoon@cs.stanford.edu

Hamilton Ulmer
Department of Statistics
Stanford University
hamilton.ulmer@gmail.com

## ABSTRACT

We describe a new hybrid data structure for dynamic graphs and identify a particular approach to visualizing it. The sample data in this case comes from a related project involving the logging and analysis of Twitter data originating from real-world conference communication on that network as discoverable via pre-shared `#hashtags`. Our data structure (referred to as a `TimeGraph`) is an abstraction that accepts updates regarding the state of the system and can efficiently render the system at any given time. The visualization shows using color, size, and animation the differences between any two particular temporal states of the graph. Visual insights within this particular dataset are examined, and future optimizations of the data structure and improvements to the visualization are discussed.

## 1. INTRODUCTION

Graphs are an incredibly versatile tool for modeling complicated physical and virtual systems. From family trees to automobile traffic routing, graphs consisting of a set of nodes and edges have proven themselves vital to humanity since long before the dawn of the Information Age. Graph theory is an important component in any mathematics or computer science education and much research has gone into developing useful algorithmic approaches to understanding this data structure. Additionally there exist superb toolkits and libraries that efficiently utilize much of this research. In the world today graphs are as ubiquitous as they are useful.

Graphs are highly applicable to many emerging data problems, but there is a significant gap in the standard toolchain used by analysts and researchers to capture and sift through data bound for a graph. There are highly functional and efficient toolkits for analyzing graphs, but almost uniformly they rely on graphs to be already formed, static entities. This makes the process of analysis cumbersome and limited to those individuals with the skills to process and reformat data into new structures. It also limits how fast the system can react to new data. What is sorely lacking in graph-style analysis is a simple conceptual model that encapsulates updates to the system in a modular and intuitive way. Tiresias is designed to simplify this process by distilling the process of data collection in a flexible way that is easy to tie into rapid concurrent data sources. Indeed, with constant increases in global connectedness and sweeping upgrades in existing communications networks, more and more data is becoming available in real-time. Making analytical sense of highly concurrent human endeavors should be a priority for quantitative individuals and organizations.

In terms of visualizations, much effort has been spent in pursuing novel algorithms for drawing graphs in a way that allows the human mind to extract meaningful conclusions about the data from their visual structure. Extensive work has gone into a variety of strategies, such as leveraging mathematical properties of the adjacency matrix [5] and in the application of physical principles to model virtual entities in stable energy configurations [1]. These techniques are well established and with some parameter cajoling can produce acceptable graph layouts in polynomial or better time. This field is impressive and has produced great gains, but it is not the focus of our visualization component. Instead we seek to identify the strengths and weaknesses common to all graph models when presented with the dimension of time. In this paper we will present some fundamental techniques (already implemented in the Tiresias viewer) and discuss future improvements in both data presentation and user interactivity that may enable increased insight into datasets leveraging the TimeGraph structure.

## 2. RELATED WORK

Dynamic graphs have a significant presence in the field of computer networking in the form of so-called "streaming graphs" where edge weights change to reflect volatile network state. Prominent uses of this model include both classic distance-vector routing algorithms and new work regarding efficient streaming graph metric estimation [2]. There is extensive interest in online graph metric update algorithms due to the difficulties inherent in storing and processing some especially high-throughput data sources. In these cases what is of prime importance is the possession of an accurate up-to-date picture of the system. Less important for these researches is the ability to accurately remember the state of the graph at any time. This feature would be very useful in an analysis context but would likely impact performance in the high-throughput applications envisioned for these algorithms.

Leskovec et al. also take a more predictive approach in their research on time-based graphs [3]. Seeking to model and understand the growth of real-world networks, they astutely observe that "the bulk of prior work on the study of network datasets has focused on static graphs, identifying patterns in a single snapshot, or a small number of network snapshots." Without ever dipping too heavily into implementation, they go on to describe their collection and observation of specific power-law densification properties and various attachment models that approximate these power-law degree distribution. One gets the sense in reading this work that a standardized `TimeGraph` data structure may have been of use in their analysis, but many of their datasets are "pre-cooked" snapshots taken at discrete intervals. Their models would only be improved by more data points, which, if sourced from a `TimeGraph` walked forward event by event would comprise the optimal approximation of continuous graph evolution.

In terms of visualization, highly meaningful related work was conducted by Veldhuizen in his Ubigraph system [6]. Ubigraph represents a system where a physics-based world is used to visualize a finite graph in three dimensions. It is visually compelling to watch the system's force-directed model evolve in adapting to changes in graph structure. Veldhuizen himself is aware of the implications of such dynamic visualizations when he describes his efforts to understand the growth of complex algorithms, ad hoc wireless networks, databases, distributed systems, and even visual performance profiling. His primary method of conveying information to the user is motion, as all of his nodes are discretely fascinating physical entities. This focus on motion, combined with the force-directed approach, works well on graphs with inherent symmetry or limited complexity, but there are cases where a built-in dependence on a specific layout system could limit the flexibility of the user in leveraging the best method for his or her data. Additionally, the current implementation does not provide a tight coupling to a time based data structure and consequently variable time control seems to be absent from the interface. Still, the speed and effectiveness of its continuous force-directed layout and the physical intuition about graphs it enables should be important considerations for any such system.

# 3. DESCRIPTION OF SYSTEM

## 3.1 Model Side

### *3.1.1 Update*

The basic building block of a `TimeGraph` is the `Update` class. An `Update` represents any change that can happen to a specific node in the dataset being encapsulated by a `TimeGraph`. It has the following interface (here simplified and expressed in pseudocode):

```
class Update:
    constructor(node_id, u_type, content, timestamp)
    rehydrate(serialized)
    serialized()
```

Standard types of `Update` objects are constructed for events that influence graph structure. In this case the update type (`u_type`) could be `+present` denoting that at the time specified in the `timestamp` field the node became present in the graph, and before that it was not. Another example is the `+edge` update, which indicates that the content field contains a list of edges that were added to the graph at the specified time.

In our implementation, `Update` objects are capable of a very useful "self-serializing" behavior whereby they can convert themselves into byte-encoded strings that can then be easily passed via the network or the filesystem. Constructing an empty `Update` and then calling `rehydrate()` with this string quickly rebuilds the update. This makes the creation and propagation of `Update` objects an extremely flexible process and as we shall see this self-serializing programming pattern is used to great lengths in many Tiresias objects.

### *3.1.2 TimeNode*

The `TimeNode` is a class that represents one node entity in the graph as it travels through the data timeline. It has essentially the following interface:

```
class TimeNode:
    constructor(id)
    archived()
    reinflate(archived)

    is_present()
    out_edges()
    in_edges()

    get_cur_time()
    get_range()
    render(time)
    add_update(update)
```

The first group of methods applies to the construction and storage of `TimeNode` objects. Every `TimeNode` must have a unique id that is used as a global key for it throughout the system. Self-serializing is also used here to allow the `TimeNode` to simplify itself into a hashmap for saving to the filesystem or appending as metadata to the NetworkX graphs generated by the `TimeGraph` class.

The next group contains the traditional adjacency interface for graph connectivity. Calling these methods yields the out-edges and in-edges of this node at its current rendered time. Note that because the `TimeNode` abstracts a node over its entire lifetime with the larger graph the node need not be present in the graph at a specific time. The `TimeNode` knows this because of the `+/- present` updates it receives and intuitively yields no out-edges or in-edges when the node is not present.

The final group contains the mechanism for updating the state of the node entity. The `TimeGraph` in Tiresias is node-centric, meaning that all updates to graph structure are stored directly in the nodes they affect. The `TimeNode` stores a vector of all the updates related to it ordered via their timestamps. Since these updates are atomic, rendering a node to its state at given time can be quickly accomplished

by rewinding and fast-forwarding the record and pushing/popping the corresponding `Update` objects.

### 3.1.3 TimeGraph

The heart of Tiresias on the model side is the `TimeGraph` data structure. Simply put, the `TimeGraph` is an abstraction encapsulating the entire timeline of the graph system it models. It has the following interface:

```
class TimeDiGraph:
    render(time)
    get_graph()
    get_cur_time()
    get_range()

    set_filename(filename)
    save()
    load()
    set_extra(obj)
    draw(filename)

    add_update(id, update)
    cache_node(node)
```

The first group of methods contains the primary time manipulation functionality. The user of the `TimeGraph` can call `render()` on any possible time value (in our implementation any valid Python `datetime` object). This method sets the state of the `TimeGraph` to be the configuration of the graph present at that time. In our implementation the user can then retrieve a NetworkX (the standard Python graph analysis library) object by calling `get_graph()`. Invoking `get_range()` returns a tuple of the start and end times of the records that compose the `TimeGraph`.

The second group of methods contains useful functionality for interacting with the system and the client. `TimeGraph` objects can easily be saved or loaded from a given filename via the previously outlined "self-serializing" idiom. Additionally they contain a user-supplied heterogeneous hashmap of metadata in the form of its `extra` object. This provides a location for particular applications to store additional caches or objects (like associated media content for the graph) in a safe place that will be automatically preserved with the graph itself when saved. There is also a convenient `draw()` method that will use the currently configured graph layout system (GraphViz `dot` by default) to output a static rendering of the graph at the current active time.

The last group of methods facilitates the addition of new information to the graph. The `TimeGraph` must store a `TimeNode` for every node that is ever present in the graph, and (as previously mentioned) the `TimeNode` objects themselves are the sole storehouse of update information. This means that the most sensible internal data structure for the `TimeGraph` is a cache of `TimeNodes` implemented as a hashmap. When a `TimeGraph` receives an `Update`, it fetches the relevant TimeNode in constant time from a hash on its ID and passes the node the update, which is then appended to its update vector. Rendering the graph at a specific time is thus a relatively simple process: just render each node in the cache and traverse them until every node that considers itself "present" is included in the graph and all of the edges between two "present" nodes have been placed. A few optimizations to this process have been considered, but the runtime for the naive algorithm is already $O(n + m)$ and greater constant factor performance has not currently been deemed necessary.

Tiresias also contains other data structures and additional operations and functions whose primary focus is the intuitive construction of updates to the graph. For example, `TimeNode` objects support subtraction, so it is possible to subtract an earlier `TimeNode` from a later one to find the difference between them. This difference object can then easily be converted into a series of atomic serialized update strings and sent to any other Tiresias objects.

## 3.2 Visualization Side

Tiresias currently implements a viewer program for visualizing the `TimeGraph` data structure. The program uses OpenGL to create an interactive hardware-accelerated display of a given `TimeGraph` or relevant subclass.

### 3.2.1 Display and Interaction

By default viewer uses an orthographic top-down projection to convert the three-dimensional space of OpenGL to the two-dimensional space where graphs are commonly drawn. When opened on a particular saved `TimeGraph` the viewer displays a visual rendering of the earliest state of the graph. The user can drag the graph to pan and use the mouse wheel to perform a familiar adaptive zoom. In addition, the user can use the right mouse button to draw a bounding rectangle on the screen that rescales the window when the button is released.

### 3.2.2 Time Differentials with Keyframes

The viewer supports interactivity through time via the use of keyframes. Tiresias automatically splits up the range of any `TimeGraph` into (by default) 60 evenly spaced regions and designates each divider as a discrete keyframe point within the otherwise continuous time dimension. The `TimeGraph` is rendered and placed at that time and the resulting layout information is saved in the `TimeGraph`'s metadata dictionary. When the user opens the viewer and the initial state is rendered, a slider bar appears on the bottom of the screen. Dragging the knob shows the user the timestamp of the keyframe assigned to that region of time. Releasing the knob initiates a visual transition between the current time and the selected time.

### 3.2.3 Visualizing the Difference Between Keyframes

In relation to these two keyframes, any given node or edge can be described as "departing" if it belongs in the old but not the new, "arriving" it belongs in the new but not the old, and "staying" if it belongs in both. Transitioning between keyframes lasts for a specific time period and currently involves visual queues derived from these classifications conveyed on three channels: motion, size, and color. In the current implementation edges are not animated or drawn during transitions to keep the screen uncluttered and to maintain a high framerate. Information about edges is thus (with the

exception of "departing edges") currently only provided by color and size channels before and after the transition.

With regards to motion, nodes in the "arriving" set appear at the origin and are tweened to fly to their eventual destination by the end of the transition period. Since all of them start at the same location and quickly diverge, the user quickly learns where to look for new nodes and can quickly gauge the number to be added by the scattering effect of the "flock". Nodes in the "departing" set are similarly animated moving towards the origin, and watching the two sets (which are easy to distinguish by their color) pass by each other on their paths into and out of of the graph gives a good sense of the scale and distribution of the nodes that are changing. Nodes in the "staying" set are simply shifted to their positions in the new layout.

Tiresias also exploits the property of the default GraphViz layout engine to maintain uniform node density to leverage the user's perception of size in comparing two graph times. With near-uniform node density an important part of the layout heuristic (quite sensibly to avoid knots and gridlock), more nodes entering the graph means a larger graph. In the Tiresias viewer this translates to overall graph expansion and contraction patterns that correlate with the number of nodes in the current keyframe. Since the user's position in space does not change during transitions, a larger, more complicated graph leads to a larger virtual world in the viewer. This property exploits our innate human sense of scale and literally immerses the user in the "big picture".

Finally, color plays a large role in the viewer's transmission of information. Green in applied to the "arriving" set, red is applied the the "departing" set and blue and gray are applied to the "staying" set. More specifically, when new nodes are joining the graph they appear drawn in green and they (along with the new edges) stay green until time shifts to another keyframe. During the transition, nodes that are leaving the graph turn red as they move off the graph. In addition, when the edges disappear during animation, edges that are being removed from the graph persist for the duration of the animation and turn red. This scheme effectively highlights the important changes taking place to the user using familiar colors: green is the color of new plant growth and red pens are commonly used to make subtractive revisions to documents. The only limitation of this color scheme could be its confusion to individuals with red-green colorblindness–the most common form.

By utilizing motion, size, and color, our initial implementation is able to clearly show changes in a highly complicated system that enables the user to draw interesting conclusions about a previously unexplored dimension in his or her data.

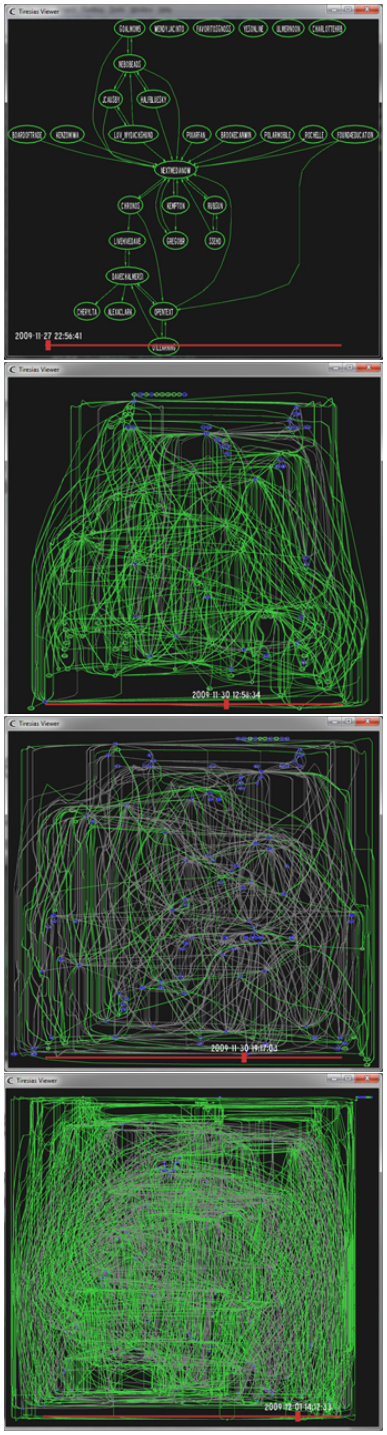

**Figure 1: The evolution of the nextMEDIA '09 conference as viewed on Twitter. Membership in the graph indicates the user has used the `#nextmedia` hashtag in a post. The edges are "following" relationships. At top you can see an initial interest group that has talked about the conference a few days before it started. Moving down we can see a large bump as all of the attendees announce their arrival in Toronto. In the later screenshots we can see sporadic green edges, indicating that new relationships between individuals are being formed at the conference.**

# 4. DISCUSSION OF DESIGN

## 4.1 Data structure performance

In several weeks of testing, the data structures in Tiresias have been performing very well. Much of this is due to the design which allows most common operations to run in linear or better time. For example, the following is the procedure for subtracting two `TimeNode` objects:

```
define subtract_nodes(a,b):
    a_in_edges = set(a.in_edges)
    b_in_edges = set(b.in_edges)
    a_out_edges = set(a.out_edges)
    b_out_edges = set(b.out_edges)

    a_presence_change = None
    if a.present and not b.present:
        a_presence_change  = '+present'
    else if not a.present and b.present:
        a_presence_change = '-present'

    a_in_edges_gained = a_in_edges - b_in_edges
    a_in_edges_lost = b_in_edges - a_in_edges
    a_out_edges_gained = a_out_edges - b_out_edges
    a_out_edges_lost = b_out_edges - a_out_edges
```

While not terribly complicated, this example shows that with good internal data structures (like no-nonsense Python hashsets), `TimeGraph`s should easily scale to many thousands of nodes.

## 4.2 Automatic consistency checks

In deployment situations there is a rather large side benefit to piping all relevant incoming data into one or more Tiresias instance: automatic consistency/sanity checking. As opposed to a standard plaintext or relational log, with "autorender" mode activated Tiresias actively generates and reassembles a model of the outside world (in the form of a `TimeGraph`) in real-time. Because of this, Tiresias can spot irregularities or inconsistencies (malicious or otherwise) in the incoming data. For example, if Tiresias is modeling a large corporate network and starts receiving `https` traffic updates for a `TimeNode` representing a previously liquidated machine, Tiresias' automatic inconsistency reporter will alert the user and prevent a potential data burglary attempt. Tiresias makes it easy to model large systems as a graph, and in doing so imbues a chaotic concurrent data ecosystem with a fundamental intelligence. This yields to the user a system aware of its own behavior via logical compliance with the model.

## 4.3 OpenGL

In our previous visualization projects[4], we have learned the importance of using the right tools when dealing with potentially massive datasets. Even a cursory examination of hardware performance trends from the past decade should yield astonishment at the progress of discrete Graphics Processing Units (GPUs). With performance numbers in consumer-grade GPUs on the order of teraflops, utilizing these devices should be an imperative for any developer at all concerned with visualization performance. For interactively rendering hundreds of nodes and thousands of edges, using OpenGL (even for an orthographic projection of two dimensional data) simply outclasses conventional framebuffer canvases. On top of this, modern cards provide hardware anti-aliasing. In future refinements we hope to exploit the programmable nature of modern pipelines (vis-ï£¡-vis programmable vertex and pixel shaders) to enhance our use of color and motion in the visualization at little or no performance penalty.

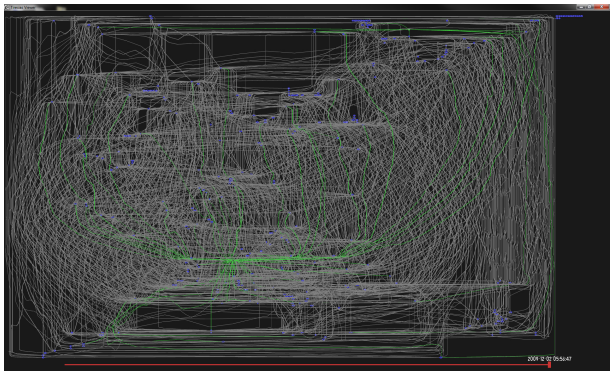

**Figure 2: The final state of the `#nextmedia` conference (a graph with 308 nodes and 2448 edges). This can be rendered and interacted with at a resolution of 1920x1160 with an acceptable framerate on an AMD Radeon 4850 (a midlevel consumer card from early 2008). In fact, the system is likely CPU-limited due to the overhead of display logic in Python. The raw performance of GPUs running 3D APIs (even for 2D projections) is clearly the future of intensive data visualization.**

## 4.4 Blackboxing layout

### *4.4.1 Why not concurrently develop a layout engine?*

Tiresias as a whole is largely concerned with the initial effects of including time in a graph data structure and in derived visualizations. As a result, we saw fit to export the task of laying out specific keyframes to the extremely popular GraphViz library. By and large, graph drawing is nowhere near a "solved problem", so we allowed the user to select the system best suited to his or her dataset. In the absence of a "magic bullet" for laying out arbitrarily complex graphs, we opted to instead include a satisfactory default and a parsing system for converting the output of many other layout systems to OpenGL.

### *4.4.2 Parsing XDOT into OpenGL Primitives*

GraphViz's `xdot` file format is an extension on its `dot` format to include spatial layout data as attributes in the existing list of nodes and edges. The header defines a bounding box and vector data in each node's line of the file has both an $(x, y)$ coordinate pair and an $(w, h)$ tuple sufficient for the standard form of an ellipse, $(x-w)^2+(y-h)^2 = 1$. For edges, we are given vertices for the arrow if the edge is directed and a series of control points for a B-spline. The B-spline can be easily rehydrated into line segments for a curve of arbitrary smoothness via multiple passes of the following subdivision algorithm:

```
def subdivide_bspline(flat_verts):
  passes = 2
  sub = [(flat_verts[2*i],flat_verts[2*i+1])
            for i in xrange(len(flat_verts)/2)]
  n = len(sub)-1
  for j in xrange(1,passes+1):
    old = sub
    sub = [(0.0,0.0) for i in xrange((2**j)*n+1)]
    for i in xrange(len(old)):
```

```
    #make even point
    if i-1 >= 0 and i+1 < len(old):
    #interior points
      sub[2*i] = ( 0.125*(old[i-1][0] +
                   6.0*old[i][0] +
                   old[i+1][0])
                   ,
                   0.125*(old[i-1][1] +
                   6.0*old[i][1] +
                   old[i+1][1]) )
    else: #end point case
      sub[2*i] = old[i]

    #make odd point
    if i+1 < len(old):
      sub[2*i+1] = ( 0.125*(4.0*old[i][0] +
                     4.0*old[i+1][0])
                     ,
                     0.125*(4.0*old[i][1] +
                     4.0*old[i+1][1]) )
return sub
```

Creating the ellipses for the nodes in OpenGL is a trivial loop around the ellipse using some basic trigonometry.

# 5. APPLICATION EXAMPLE: SOCIAL AVIARY

One successful deployment of Tiresias was done concurrently with its development. The deployment, referred to as "Social Aviary" was conceived of by Hamilton Ulmer (a MS student in Stanford Statistics) as a project for CS322: Network Analysis. The idea comes from a desire to witness and model interactions between individuals on a social network that is directly tied to a real event. In essence, Social Aviary aims to use Twitter as a raw data source for studying and predicting emergent social dynamics in groups.

## 5.1 Tracking a New Community

A specific Twitter subgraph in Social Aviary is a `TimeGraph` based around a specific `#hashtag`. A Twitter `#hashtag` (like `#python` or `#stanford`) is a notation that can be used in a Twitter post to denote its content and link it to a specific topic. Conference organizers have begun the practice of pre-sharing a tag for their conferences to promote a public discourse and encourage networking. We took advantage of this usage pattern to get a sense of group dynamics and network evolution. The TimeGraph for Social Aviary has users represented by `TimeNode` objects and directed edges that represent the Twitter relationship of "following" (where every post by the person you're following is shown on your homepage). Formally, at any given time $t$, the graph of the system contains nodes for all users who have used the conference `#hashtag` at a time less than or equal to $t$. At times we have jocularly referred to this structure as a "twit mob".

## 5.2 Implementation

Social Aviary was implemented using Tiresias in the configuration shown in Figure 3. In this architecture, a central "update publisher" server was written for our local network. This program, composed of about 30 threads of various types, had the task of maintaining a single "watch list" of users and search terms that it would broadcast `Update` objects for via PYRO[1]. This was accomplished satisfactorily for a large number of individuals via simultaneous usage of all three Twitter APIs. When the server starts up, it creates for each user a `TimeNode` derivative and pulls the current state of the user into it via the web. This "initial pull" is not broadcast as any client would be expected to pull that data on startup while the publisher is more concerned with live updates.

With the current versions of each user in hand, the server enqueues them to be reexamined via the REST API for follower/followee changes. With these connectivity updates (which currently must be obtained via polling), the publisher schedules them to be reexamined by a worker thread in a few minutes ($\pm 25\%$ random jitter to avoid pathological congestion). During re-examination a new fresh copy of the User is pulled from the web and has the old copy subtracted from it via the previous mentioned procedure of `TimeNode` differencing. The resulting `TimeDiff` object supports being broken down into its component update types, which are then converted to byte strings via the aforementioned "self-serialization" idiom and can be broadcast accross the LAN to any and all researchers who have subscribed to this node's ID.

For the Streaming API the process is more direct, although the limitations of this new API require some trickiness. Whenever an item is added or removed to the watchlist the publisher creates a persistent Streaming API connection objection with a callback option to report on any new posts by those users or posts contain one of our interested search terms (i.e. the conference `#hashtags`). Whenever we are called back we simply construct the `Update` and enqueue in the publisher's broadcast queue. Two problems with the Streaming API make life significantly more complicated. First, connections silently go stale after an unknown period, so we leverage a TTL (time to live) field inside the Streamer object so that it can be destroyed and remade before this happens. Secondly, currently the streaming API is limited to following the new posts of at most 400 users. At the peak of the two conferences we tracked our watchlist numbered over 800 users. To mitigate this limitation, during TTL-initiated streamer reconstruction we simply select a random 400 user subset of our entire watchlist. The random slices being altered ever few minutes means we have good coverage, and we need not miss any updates because they would get pulled during the next REST API call. The only penalty is a statistically significant expected delay increase in notification time.

Lastly, the Search API is only used on client resume when it has to check for posts matching its `#hashtag` that occurred while logging was deactivated. To do this quickly we simply generate a few simultaneous calls for the top few pages of the search results. We then perform hashset difference on the owners of the new posts with the owners we know about from before. New participants are added to the watchlist and a new node is initialized in the `TimeGraph` node cache. When visualized to the current time, the system shows the emergence of new green nodes and edges flying into the graph.

[1]Python Remote Objects (PYRO), created by Irmen de Jong. `http://pyro.sourceforge.net`

## 5.3 Results

The findings of the project are somewhat complicated and are beyond the scope of this overview, but let it suffice to say that after a few hiccups involving inconsistencies within the Twitter API and unicode bugs Tiresias was able to provide a solid and accurate record of our new `#hashtag` based subgraphs. We successfully logged a week's worth of data on both the Online Information/IMS 2009 Expo (`#online09`) and the Toronto NextMEDIA conference (`#nextmedia`). Both of these conferences deal with "new media", so the concentration of active Twitterers at the conferences was very high. At our peak, we logged more than 800 users simultaneously. We were able to do this comfortably with a ten minute base refresh time on connectivity polling. A finer resolution would have been nice, but even with the research whitelist we obtained from Twitter we were limited to 20k REST API queries per hour. At two queries per pull, plus the overhead of initial pulls at startup, there is not too much room to maneuver in the 20k limit. We managed to shunt as much data collection as possible to the streaming process which are not rate limited in the same way. These frees our REST calls for data we cannot get anywhere else.

The data itself was very interested when viewed in the interactive visualization. Figure 1 shows a few screenshots of the evolution of `#nextmedia`. It is interesting to read the stream of excited posts as a conference starts to heat up while watching the graph itself expand very quickly. Conferences like the ones we model tend to have a core interested group (perhaps planners/employees of the organizations that conduct them) that exists long before the conference starts. The graph simply explodes the day before the conference, and in the process of the conference the graph really does become more cohesive. Using Tiresias helps us to see that interaction on Twitter really can mirror real-life events in some cases.

# 6. FUTURE WORK

We have many ideas on how to improve Tiresias moving forward.

**Time-rendering optimizations** Several optimizations are possible with regards to rendering the state of the graph at a given time. For a `TimeNode`, if we maintain a dirty flag and allow old updates to be inserted and sorted into the array we can use a smart rewind/fast-forward system to avoid almost ever having to move through the entire update array. Additionally, we are contemplating a callback function supplied to each node during rendering that would allow the graph object to update along with the node, largely saving us the extra step of re-traversing it at the end. These optimizations, if perfected, could help us support model side graphs on the order of millions of nodes and edges.

**Domain-specific metrics** We are currently considering an API for allowing the client a standardized way of supplying new metrics to the visualizer. Such an API would allow the client to tie visual attributes like size and color to metrics within his or her own `TimeGraph` object. An example of a metrics plugin for Twitter would be a coloring mode that uses a statistical profile of spammers to color the graph by likelihood of being

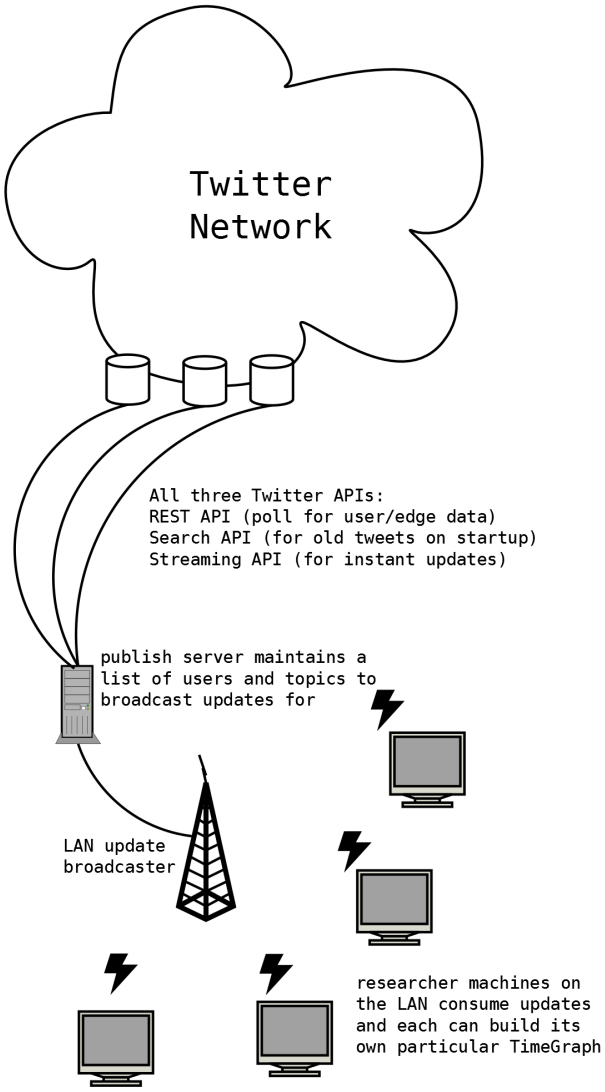


**Figure 3: An application of Tiresias to log extensive sub-networks on Twitter. One persistent publisher server (cleared by Twitter for the API research whitelist) maintains a watchlist of the individual researchers' `TimeNode` components and can alert them when they should expand their graphs. The publisher-subscriber pattern makes this scalable, as the watchlist can be divided between additional publishers on the LAN without any modification to the researcher clients. Performance in this configuration is very good, and parameters can be tuned to make optimum use of Twitter API policy.**

a spammer. The basics of such metrics are already implemented (by pressing the space bar in the viewer), but an intuitive API would allow for optimal coupling.

**Intuitive real-time visualization** Real-time visualization, while not currently supported, was designed for during development. New keyframes can easily be added to a `TimeGraph`, and an automatic "play button" on the visualizer would show the evolution of these graphs in real time. The main impediment to this in the current configuration is the use of GraphViz's "dot" layout engine. For graphs on the order of 200 nodes and 1500 edges, "dot" can take many minutes to lay out the graph. Making it easier for the user to configure a layout engine perfect for his or her task is a goal moving forward, and progress on this goal should help make real-time visualization a no-brainer.

**New interaction techniques** Interacting with graphs is still a very new phenomenon. Clicking and dragging and the mouse wheel zoom are a good start but there should be a lot more fertile territory to explore. Some of this exploration should revolve around common graph algorithms. For example, it should be intuitive to select a series of nodes and see the shortest path or minimal spanning tree involving them. Also, being able to click a node and roll the mouse wheel to highlight each level of the connected set would go a long way towards helping the user make sense of a complicated graph. Lastly, allowing the user to maintain a specific "interest set" to observe through time would be a powerful "role-playing" tool for building intuition about a particular system's functioning. The intersection of social networks, graph theory, and Human-Computer Interaction makes this task in particular an exciting avenue of useful discovery.

# 7. CONCLUSION

With how much of our increasingly data-centric world can be expressed using a time-sensitive graph it seems rather shocking that there have not been more academic resources expended in the pursuit of modeling and visualizing their uniquely useful properties. Tiresias exists merely as a first step in that direction. The rise of highly-concurrent computers capable of immense visual detail makes today's technology highly apt for the type of analysis that Tiresias aims to achieve. On top of this, concurrent advances in social and communications networks, graph analysis and machine learning offer us the opportunity to learn much more information from many more interesting datasets. Tiresias offers both an efficient, easy-to-deploy data structure and an intuitive, adaptable visualization as accessible building blocks for future research and analysis.

# 8. REFERENCES


[1] Tim Dwyer and Yehuda Koren. Dig-CoLA: Directed Graph Layout through Constrained Energy Minimization. *INFOVIS*, 2005.

[2] Sumit Ganguly and Barna Saha. On Estimating Path Aggregates over Streaming Graphs. *Proceedings of The 17th International Symposium on Algorithms and Computation (ISAAC)*, 2006.

[3] Jure Leskovec, Jon Kleinberg, and Christos Foloutsos. Graph Evolution: Densification and Shrinking Diameters. *ACM Transactions on Knowledge Discovery from Data*, 1, March 2007.

[4] Ryan Noon, Matt Jones, and Rohan Puranik. CERES: A new approach to data visualization and exploration leveraging the power of Python and modern GPUs. *Stanford University, CS194 Software Faire*, 2009.

[5] Daniel Fleischer Ulrik Brandes and Thomas Puppe. Dynamic Spectral Layout of Small Worlds. *J. Graph Algorithms Appl.*, 2007.

[6] Todd L. Veldhuizen. Dynamic Multilevel Graph Visualization. December 2007.