LASER INTERFEROMETER GRAVITATIONAL WAVE OBSERVATORY

- LIGO -

CALIFORNIA INSTITUTE OF TECHNOLOGY

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

Document Type	**LIGO-T1400365-v5**	2014/08/13

GraceDB: A Gravitational Wave Candidate Event Database

Brian Moe, Patrick Brady, Branson Stephens, Erik Katsavounidis, Roy Williams, Fan Zhang et al.

Distribution of this draft:

LIGO Scientific Collaboration

California Institute of Technology
LIGO Project - MS 51-33
Pasadena CA 91125
Phone (626) 395-2129
Fax (626) 304-9834
E-mail: info@ligo.caltech.edu

Massachusetts Institute of Technology
LIGO Project - MS NW22-295
Cambridge, MA 01239
Phone (617) 253-4824
Fax (617) 253-7014
E-mail: info@ligo.mit.edu

WWW: http://www.ligo.caltech.edu/

Processed with LaTeX on 2014/08/13

Abstract

The gravitational wave candidate event database (GraceDB) is a webservice that organizes candidate events from gravitational-wave searches and provides an environment to record information about follow-ups. This document describes the basic features of GraceDB, including its data models, controller methods, and rendering templates. In the advanced detector era, GraceDB is intended to act as a hub for communication between gravitational wave and electromagnetic observers. This imposes several additional requirements on GraceDB which will inform continued development of this service.

1 Introduction

The Gravitational Wave Candidate Event Database (GraceDB) is a webservice for collecting, storing, and presenting information about astrophysical transients, especially candidate gravitational wave (GW) events. This system is designed to aid in the GW data analysis effort and to facilitate multi-messenger studies of specific transient events. GraceDB is currently under development by the LIGO Scientific Collaboration on behalf of the LIGO/Virgo Community.

GraceDB is built on Django [1], a Python-based, model-view-controller (MVC) style web framework. In addition to the webservice itself, two client tools are provided: a simple command-line client and an example REST client (both written in Python). The latter can be used out-of-the-box or extended by users for custom behavior. Both client tools expose all of GraceDB's functionality, including new event creation, complex searches, and event annotation.

The structure of this document is as follows: The first section details the basic features of GraceDB server code. The second outlines additional requirements expected in the advanced detector era. The conclusions touch on some possible future directions for GraceDB development. Finally, the usage of the GraceDB client tools is described in the appendices.

2 Basic features

The objects of interested in GraceDB—namely, the candidate events and their annotations—are represented as structured information in a database. The webservice queries, modifies, and constructs views of this information *dynamically* in response to requests made by users. In this section, we first describe the underlying database models. Next, we describe the logic functions ("controller methods") that manipulate this information. Finally, we describe the different available representations ("views") of the data offered by the webservice, primarily HTML and JSON.

2.1 Data models

There are two broad categories of data models in GraceDB: 1) the *event* model itself and 2) the *event annotations*. The former represents the individual candidate GW event, whereas the latter models represent information added to the event *subsequent to its creation* by a human or robotic actor. The event itself is produced by a GW data analysis pipeline and is ordinarily submitted to GraceDB by a robot associated with that pipeline. After the initial event is created, automated followup robots as well as human analysts may make further contributions by *annotating* the event. The full set of information about an event, including all of its annotations, is sometimes called the event's *portfolio*.

2.1.1 The Event model

The Event model has the following attributes:

- GraceID
- time of creation
- event submitter
- analysis group
- analysis type
- GPS time of the detection
- list of participating detectors
- false alarm rate (FAR) of the detection
- likelihood

The first five items above are administrative in nature, while the last four convey physical information. Most critically, the *GraceID* serves as a unique identifier for the event. Next, the time at which the event was submitted to the GraceDB server, as well as the authenticated user responsible (whether human or robotic) are recorded. The remaining two attributes of the basic event model requiring explanation are the *analysis group* and the *analysis type*.

The analysis group field indicates which working group within the LVC is responsible for the pipeline that produced the candidate event. At present, the choices for the analysis group are: CBC, Burst, Test, External, Stochastic, and Coherent. (Note that, at present, there are no GraceDB event classes appropriate for stochastic or coherent GW detection candidates. These group options exist in the database as placeholders.) The 'Test' group is for events which are submitted to GraceDB in order to test either GraceDB itself or other followup infrastructure. (Ordinarily, when the analysis pipelines are tested on simulated data, as in engineering runs, the resulting events are *not* submitted to the Test group. This is because these are tests of the pipelines themselves, rather than of GraceDB.) Finally, the 'External' group is for transient events originating from instruments outside of the LVC (such as telescopes or neutrino detectors). GraceDB already ingests GRB events from Swift and Fermi into the External group. Such EM transients are stored in GraceDB in order to facilitate GW coincidence searches.

The *analysis type* field in the event model is intended to denote which particular analysis pipeline (within a given working group) was used to make the detection. At present, the following choices are available for this field:

- CBC pipelines: LowMass, HighMass, MBTAOnline
- Burst pipelines: CWB, Ringdown, Omega, Q, X
- External pipelines: GRB
- Other: HardwareInjection

Note that, to date, not all of the above pipelines have actually submitted events to GraceDB. In fact, by convention, only pipelines which are able to produce an estimate of the FAR submit events. (GRB events are an exception.) The last event type in the list above, HardwareInjection, is reserved for unblind hardware injections that are used to check for coincidences with candidate events.

The event model described thus far applies to all GraceDB events, and hence all must have at least these basic attributes. (For External events, however, the 'FAR' and 'likelihood' fields

GRB	Inspiral	Burst
ivorn	ifos	ifos
author_ivorn	end_time	start_time
author_shortname	end_time_ns	start_time_ns
observatory_location_id	mass	duration
coord_system	mchirp	peak_time
ra	minimum_duration	peak_time_ns
dec	snr	central_freq
error_radius	false_alarm_rate	bandwidth
how_description	combined_far	amplitude
how_reference_url		snr
		confidence
		false_alarm_rate
		ligo_axis_ra
		ligo_axis_dec
		ligo_angle
		ligo_angle_sig

Table 1: Analysis-specific attributes for event subclasses.

are allowed to be null.) In most cases, however, more is known about the event at creation time than is suggested by these basic attributes. Thus the *analysis-specific attributes* are incorporated by subclassing the simple event model described above. Three such subclasses are presently defined: 1) GRB events, 2) compact binary inspiral events, and 3) burst events. See Table 1 for details of these additional attributes. The names of these attributes are taken directly from the event files uploaded at creation time.

2.1.2 Annotation models

There are three types of annotations that can be attached to any given event in GraceDB: labels, log messages, and log message tags.

Labels: These are short strings that convey high-level information about an event. They are drawn from a list of allowed labels that is stored in the database, and are applied to a specific event by creating an association between the label and event objects. The database model for a label object consists of just two fields: the label's name and it's default color (for display purposes). At present, the allowed labels are: INJ, DQV, EM_READY, LUMIN_GO, LUMIN_NO, SWIFT_GO, SWIFT_NO, cWB_r, and cWB_s. In practice, however, only the first three labels are used. The INJ label indicates that the event corresponds to a known injection, and hence should not be considered as a real candidate GW event. DQV indicates a data quality veto, and hence also rules out an event. Finally, EM_READY indicates that the event is suitable for electromagnetic followup. In all three cases, the label conveys high-level information that could not have been known at the time of event creation. A typical workflow is as follows: An event arrives in GraceDB, triggering a search of nearby times for injections. If an injection is found within a sufficiently narrow time window of the event candidate, the INJ label will be applied. Additional labels can be added to the database as needed by the GraceDB developers.

Name	Meaning
analyst_comments	comments from human analysts entered by hand
audio	uploaded audio files
background	information about the analysis pipeline's background
data_quality	data quality information
ext_coinc	comments regarding coincidence with external transients (such as GRBs)
psd	data or plots of the noise power spectral density near the event time
sig_info	information regarding the significance of the event
sky_loc	sky localization information
strain	data files or links to $h(t)$ data
tfplots	time frequency plots (such as spectrograms)

Table 2: The names and meanings of conventional tags.

Log messages: Each event in GraceDB has an associated set of log messages that contain additional information about the event. Taken in sequence, these messages completely document the event's history and, thus serve as an *audit log*. Any time a file is uploaded, a label added, or the event data are changed, a new log message is created to document the change. Users can add their own log messages to an event (in plain text or HTML) and can attach files. Thus, the event log allows unstructured information to be appended to an event. This is the principal mode of annotation in GraceDB. The log message data model contains the following attributes:

- the event to which the message refers
- time of creation
- log message submitter
- comment (the actual message text)
- filename (the name of an attachment)
- file version (the version to which this particular message corresponds)
- the number of the log message in the event's sequence

Log message tags: Tags allow an event's log messages to be categorized for purposes of presentation or indexing. For example, applying the tag `sky_loc` to a log message indicates that the message, along with its associated files, is related to sky localization. Tagging messages is useful for creating presentation templates (since related annotations can be grouped together) and for pinpointing pieces of information stored in the log messages (since log messages can be searched according to their tags). Users may create their own custom tags for special purposes, but the use of conventional tags is encouraged for common types of log messages. These common conventional tags are shown in Table 2. Tag objects have three attributes: a name (suitable for entry on the command line), a 'display name' (used for presentation purposes), and a list of links to log messages.

2.1.3 URL patterns

The objects (events and annotations) stored in the GraceDB database are exposed to users through URL patterns. Since this is most apparent with the REST interface, we start with a discussion of the REST URL patterns. An illustrative (but not exhaustive) list is given

URL pattern	Resource
/api/events/	event list
/api/events/<GraceID>	specific event
/api/events/<GraceID>/log	list of log entries for an event
/api/events/<GraceID>/log/<log #>	specific log entry
/api/events/<GraceID>/files	list of available files for an event
/api/events/<GraceID>/files/<filename>	specific file
/api/events/<GraceID>/labels	list of labels applied to an event
/api/events/<GraceID>/labels/<label name>	specific label
/api/tag	global list of tags
/api/events/<GraceID>/tag	list of tags for given event
/api/events/<GraceID>/tag/<tag name>	details of tag for this event
/api/events/<GraceID>/log/<log #>/tag	list of tags on a given log message
/api/events/<GraceID>/log/<log #>/tag/<tag name>	log entry tag detail

Table 3: REST API URL patterns for important objects. Names in angle brackets stand in for specific values. The service URL (`https://gracedb.ligo.org/`) is suppressed.

URL pattern	Resource
/events/<GraceID>	event page (HTML)
/events/voevent/<GraceID>	event (as VOEvent XML)
/events/<GraceID>/files/<filename>	specific file (download target)

Table 4: Web interface URL patterns for important objects.

in Table 3. (These URLs should be prepended with the protocol and service URL, such as `https://gracedb.ligo.org/`. Note that all URLs in the REST API begin with `/api/`.) For any given data model, the list of objects is considered to be a separate resource from the individual objects themselves. Thus, on the top level, we have the event list resource; and below that, there are the individual events. Then, for a given event, there are lists of annotation objects below that: log entries, files, labels, and tags. Beneath the list of log entries for a given event, we then have the individual log entries themselves. In this way, all of the data objects of interest are exposed for operations via HTTP verbs (GET, POST, etc.).

By contrast, the web interface has a simpler list of URL patterns (see Table 4). Only the individual event and file resources are exposed. Notice that, with the exception of files, the annotation resources (log messages, tags, and labels) are not *directly* exposed through the web interface. This is because web users interact with an event's annotation objects through widgets on that event's web page. The VOEvent URL pattern in Table 4 is an example of an alternative event views. Rather than viewing the usual HTML web page representation of an event, the user may opt to download the VOEvent in XML format.

2.2 Controller logic

In this section, we describe the different methods for interacting with the data stored in the GraceDB database. These methods are called "controller methods," since they handle all logical processing. (Confusingly, the Django documentation refers to controller methods as "views.") When a user sends a request to a target URL, a particular controller method is exe-

cuted. Many of these methods actually come in two versions: one for the REST API version and one for the web interface. The REST API was introduced later, and an attempt was made to repeat as little code as possible, but there is still more re-factoring to be done. In particular, the web-interface methods could be re-written (at least in part) as thin wrappers around the REST methods.

As mentioned in section 2.1.3, the REST interface is organized by exposing lists of objects and the individual objects underneath them as separate resources. Sending an HTTP GET request to the target URL for one of these resources simply returns a representation (the 'RE' in REST) of the resource requested. Creating a new object is done via POST to the list resource, whereas modifying an existing object is done via PUT to the individual object resource. The only other HTTP verb implemented in the GraceDB API is DELETE (which is used for tags, but not for events or event log entries). The centrality of the objects themselves in organizing the API is in keeping with the Resource Oriented Architecture (ROA) principle [2]. For example, instead of providing a URL specifically for event creation (as in the cgi-bin model), one exposes a resource (the event list) that supports POST (which has the effect of creating a new event). The controller functions in the REST API (which are really callbacks for the HTTP verbs like GET, POST, etc.) are implemented using the Django REST Framework [3]. This framework provides a remarkable Browseable API tool for navigating the resources exposed by the API and for seeing which HTTP verbs are supported by each resource (see `https://gracedb.ligo.org/apiweb/`). This feature makes the API essentially self-documenting, and is extremely useful for coding directly against the API.

2.2.1 Event creation

Whether through the REST API or through the web interface, the process of creating an event starts by capturing data POSTed to a target URL. This data is bound to an event creation form with the following fields: group, type, and event file. Next, a bare-bones event is created in the database with the appropriate group, type, submitter, and creation time. This bare-bones event is necessary in order to obtain a GraceID (which will determine the path of the event's data directory on disk). Next, the event file is written to disk in the appropriate event directory. This file is then read and parsed in order to obtain the remaining event attributes (both basic and analysis-type specific). The complete event is then saved in the database. Finally, an attempt is made to send out alerts via the LVAlert system and email. Users may configure notifications via email using the GraceDB web interface by clicking 'Options' and creating a new contact and notification. In practice, LVAlert is almost always used in favor of email notifications, so they are not discussed further in this document.

2.2.2 Annotation

Once an event has been created, information can be added to it by creating annotations. Basic descriptions of the workflow for creating the principal types of annotations are given below:

- **Log message creation:** As in the case of event creation, log entries are created by first extracting information from the POST data. For the web interface, the POST is an AJAX request initiated by a JavaScript text editing tool. For the REST interface, the annotation contents are encoded as multi-part form data and POSTed to the log message list resource. At present, the web interface does not allow for file attachments (see request [4]). The log message creation method extracts the message text, filename, file contents, and tag name (see below) from the POST data and creates a corresponding database entry. If

a file is attached, the contents are written to a versioned file with the specified name and an LVAlert message is sent to notify listeners of the uploaded file.

- **Message tagging:** A tag may be added to a log entry either when the log is created or later as a separate step. If the log entry is created via the web interface, it is automatically tagged as an 'analyst comment'. But if it is created via the REST interface or command-line client, the tag name (and display name, if desired) may be included in the POST data. In order to tag an existing log message via the web interface, an 'add' button can be clicked next to the message in the full message log. This brings up a dialog form that allows the user to tag the message with one of the commonly used ('blessed') tags or to create a new tag for the message. With the REST interface, an existing message is tagged by making an PUT request to a target URL for the given log message and tag name (see Table 3). If the tag is not already in the database, a new one will be created and the body of the PUT request will be checked for the (optional) tag display name.
- **Uploading files:** As mentioned above, there is currently no facility for uploading files via the web interface (though they are available for download). In the REST interface, however, an event's files are exposed in the usual way, with a list resource and individual file resources. The file list supports a GET request (which returns the list of files), and individual files support GET (download) and PUT (upload). (Note that this model differs from event log messages, for example, where a new object is created by POSTing to the list of objects. Here, a PUT to the individual file resource is used because, in this case, the URL of the file resource can be constructed from the GraceID and filename, and hence is *already known* to the submitter. By contrast, the submitter of a log message does not know the final URL before submitting it because the server calculates the message number.)
- **Labelling:** At present, adding a label to an event can only be done via the REST interface or the command line client (i.e., there is no way to add a label through the web interface). This is accomplished simply by making an empty PUT request to the label URL for a given event and label name (see Table 3). The PUT callback creates a new association (a "labelling") between the event and label objects. This labelling is a database object in its own right, and contains fields for the event, label, and creator (i.e., the user who applied the label). An event log message is also generated so that the new label is fully documented. Finally, an LVAlert message is sent.

2.3 View rendering

The controller methods described in the previous section receive HTTP requests, do any required processing (such as interactions with the database), and finally return HTTP responses. The final step of constructing the response itself (given the products of any logical processing) is handled by 'views' (the V in MVC), also known as 'renderers.' GraceDB uses Django templates for the purpose of rendering content as web pages. For the REST interface, the rendering is handled by the Django REST Framework [3].

For any given data object, multiple templates may be used to render it depending on the request. If there are multiple templates, the controller method handling the request will typically handle the logic to decide which template should be used. This is particularly useful for event pages. For example, if a user issues a GET request for an for a cWB burst event page, the controller method passes a burst-event-specific template to the renderer. Django templates also support inheritance, so that the burst-specific template inherits most of its content from a generic event template.

For the REST API, content is rendered as JSON by default. There are additional renderers for the browseable API and LigoLw, but more could be added as needed. The Django rest framework takes the output of such custom renderers and wraps it with the appropriate HTTP headers to create the completed response. The content negotiation proceeds by checking the HTML 'Accept' header of the request and then comparing with the available renderers. If the 'Accept' header is not specified, then the default JSON is returned. But if it is specified, an attempt will be made to find a renderer for that media type. The controller methods may specify a priority order for the available renderers, and the highest priority renderer that can return the requested media type is selected. If none is found, an error message is returned to the effect that the Accept header cannot be satisfied.

2.4 Authentication and authorization

The first step of authentication is handled outside of the GraceDB server code by the Apache [5] HTTP server. There are two modes of authentication:

- **Shibboleth:** When a user arrives at the GraceDB web site, the Shibboleth [6] service provider (SP) daemon on the GraceDB host redirects the user to the LIGO Identity Provider (IdP) with an authentication request. The user authenticates using his/her LIGO.org credential, and then is redirected back to GraceDB with a valid IdP session cookie. Apache's `mod_shib` processes the cookie and populates the `REMOTE_USER` and `isMemberOf` environment variables. From GraceDB's point of view, the existence of a `REMOTE_USER` value indicates that a user has successfully authenticated, and the `isMemberOf` value indicates his/her group memberships. Next, Django adds both of these attributes to the `request` object, where they can be accessed by the auth middleware and controller methods.
- **Certificate-based auth:** The REST API is protected with certificate-based authentication, which is handled by Apache's `mod_ssl`. The user is required to present a valid X509 certificate (or proxy) that traces back to a trusted CA. Then the environment variables `SSL_CLIENT_S_DN` (the client subject's distinguished name, DN) and `SSL_CLIENT_I_DN` (certificate issuer DN) are populated, and Django adds these attributes to the request object.

The GraceDB webservice maintains its own database of users. When a request comes in, the auth middleware searches this database by username (the eduPersonPrincipalName, obtained from `REMOTE_USER`) if the user requested a Shibboleth-protected URL, or by X509 subject DN if the user requested a REST API URL. If the search turns up a user, Django sets the 'user' attribute of the request object, and processing of the request continues.

But suppose no corresponding user is found. If the original request came with a valid LIGO IdP session cookie, we can be sure that this is a legitimate user. Thus, a new user object is created in the GraceDB local user database, and the processing of the request continues. However, if the user authenticated with an X509 certificate and is not found in the database, a 'Forbidden' response is returned. This is because we cannot be sure that this X509 certificate corresponds with any valid LIGO.org user.

Because the X509 subject DNs must be found in GraceDB's user database, the database is periodically updated using the LIGO central LDAP. This method works for all human users. The DNs of robotic users are entered into the database manually, along with an appropriate user name and contact information. This way, robots may authenticate to the REST API and perform operations, and their actions are logged in a recognizable way.

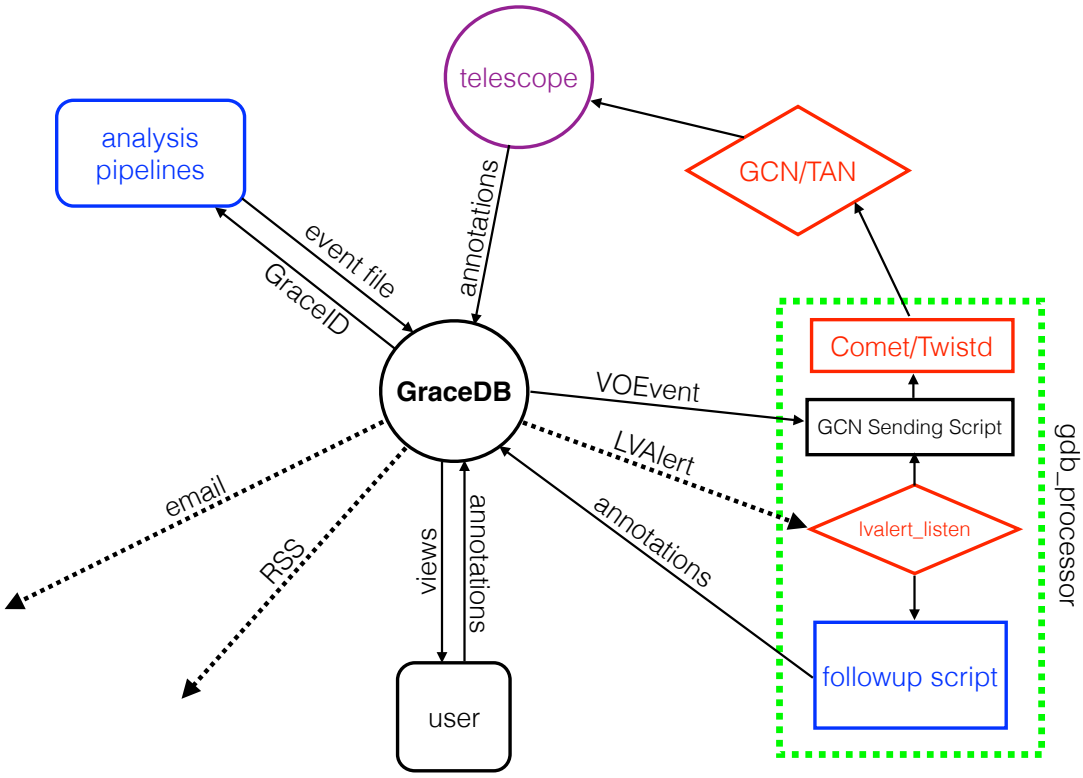

Figure 1: Some example workflows, starting from the GW analysis pipelines and ending with annotations provided by EM observers.

2.5 Summary

The foregoing sections present a sketch of the current state of GraceDB. We have described the resources exposed by GraceDB and the operations that can be performed on them. One of the key architectural choices made by the GraceDB developers was that GraceDB should not generate content, but rather should archive content from external sources (namely, from LVC data analyses and follow-up processes) and should present this content in a useful way. Thus, GraceDB is not itself responsible for doing any follow-up analyses of candidate GW events. Instead, such analyses are often performed by an automated user (the `gdb_processor` robot) in response to events and annotations arriving in GraceDB. Figure 1 illustrates this interaction, along with some other possible workflows involving GraceDB.

3 Additional requirements

Some additional requirements for the advanced detector era are outlined in this section and entail several challenging development tasks. Most of these requirements are aimed at facilitating multi-messenger astronomy.

3.1 The electromagnetic followup bulletin board

The electromagnetic followup bulletin board (EMBB) is a proposed space within each GraceDB event page for gathering and displaying information related to EM followup. We anticipate that at least three general categories of annotations may arise from the EM followup effort:

- **observation records** documenting individual observations (i.e., patches of sky that have been imaged, with accompanying details)
- **candidate source records** documenting sources discovered in the course of EM followup observations (i.e., potential EM counterparts)
- **unstructured comments** containing any additional information MOU partners may wish to share (free text)

These different types of annotations will vary in the amount of *structured* information (i.e., information that is representable in the database) that they contain: from none (free text comments) to almost all (observation records). So a flexible data model will be required. Furthermore, it is not known at present how the electromagnetic astronomy MOU partners will prefer to report on followup observations and candidate EM counterparts. These facts recommend a relatively light-weight data model for EMBB annotations.

The current plan is to create a new data model within GraceDB for an EMBB log message: EMBBEventLog. This will be a general purpose model for all EM followup related data. Thus, the model should be rich enough to adequately represent the three categories of annotations mentioned above, but simple enough to keep the workflow manageable. Some requirements on this model are outlined below.

> **Requirement 3.1.1.** The EMBB should provide a mechanism for astronomers to record observations in a structured, searchable manner. This information should be accessible through the web interface as well as through the REST API in a machine-readable format.

The EMBBEventLog messages that document particular observations should contain several pieces of structured information, including: the footprint of the observation on the sky, the waveband, and the provenance. For the footprint, our proposal is the 'smallest covering rectangle', defined by a center and the width in the RA and Dec directions: for an observational footprint, this is the smallest rectangle containing the imaged area; and for a candidate source, the width measures would be the position error. Thus the physical location of the rectangle can be defined by the four dimensions of sky position, time, and EM waveband:

- RA and Dec of the center of the equatorially-oriented rectangle in decimal degrees, with the widths in RA and Dec of the rectangle, also in decimal degrees.
- The GPS time of the imaging time, with the duration of the imaging in seconds.
- The waveband of the imaging, expressed in general terms by a vocabulary of wavebands.

A suggestion for the waveband vocabulary is to use the 'Unified Content Descriptor' language [7], an international standard from the virtual observatory. Some examples are: `em.opt.R` (Optical band between 600 and 750 nm) and `em.X-ray.hard` (Hard X-ray, 12-120 keV).

We will want each EMBBEventLog to be adequately identified and provenanced both to the GraceDB system and to the EM followup observers:

- The facility that took the data, chosen from a list derived from the MOU list.

- The name of the responsible scientist who is submitting the EMBBEventLog.
- The facility's unique identifier for the observation, chosen as they wish.

Then by combining the identifier for the facility with their own identifier, we have a unique identifier over all submissions from all facilities.

In addition to the structured data outlined above, we also want to allow unstructured information to be added to the bulletin board:

> **Requirement 3.1.2.** The EMBB should provide a mechanism for astronomers to annotate events with unstructured information (e.g., freely composed text and images). These annotations should be displayed in the EMBB web area.

As an intermediate between the structured data fields and the text field provided for freely composed comments, we should also support arbitrary machine-readable metadata in the EMBBEventLog. This would likely take the form of a serialzed dictionary of key-value pairs, chosen by the observers themselves. Rather than the endless task of making a list of all possible keywords (for example `{"limiting-magnitude" : 22.3}`), there can be a single place for this formal text.

> **Requirement 3.1.3.** The EMBB should provide a mechanism for astronomers to annotate events with arbitrary formal information (e.g., JSON, XML, etc.). These annotations may be understood by clients of the EMBB, even if the parameters are not yet known to GraceDB.

As noted, it may be that data is put into this formal text that GraceDB does not understand; that astronomers may be creative with keywords in representing the data. However, the important matter at the start is for this formal text field to be sufficiently flexible, so that standardization and schema-making can come later with experience.

There is also a need for flags to properly identify the meaning of the EMBBEventLogs: it may be a footprint or a candidate source, it may be a completed observation with completed data analysis, or one that will happen in the future, or it may be just an output from a planning program (a possible observation).

It is likely that information related to followup observations and potential EM counterparts will appear in content sources external to GraceDB. Thus,

> **Requirement 3.1.4.** The EMBB should incorporate and/or link to information about an event from other content sources (such as GCN/TAN Circulars, or even richer communications like GCN reports, if the EM astronomers choose to use them.).

If this information is a mixture of formal (i.e., machine-readable) and informal (free text), then the formal part should be ingested into the formal part of the EMBBEventLog, and respectively for the free text.

The EMBB should be able to take inputs in various ways: from filling in a web form or from sending an email, and of course it can be either humans or machines taking these actions. One way to allow structured input via email is to extend the syntax of the GCN Circular, so that those astronomers who are used to emailing Circulars can do so for their MOU-required observation reporting.

The EMBB should be able to support client applications that can show the information in various ways, or fuse it with other content. Queries for EMBB data would generate formal responses (JSON or XML) in the standard ways: fetch a list of items, fetch an item in detail, edit an item, and so on.

3.2 Presentation and searching of skymaps

GraceDB events may be searched by any of the fields in the event data model (including analysis-specific attributes), but there is not yet any way of searching by sky location. Including such a "cone search" facility in GraceDB could aid in coincidence searches (both automatic and human-initiated).

Requirement 3.2.1. GraceDB events must be searchable by sky location.

The details of the sky-location queries that should be supported are still under discussion. The queries could resemble, for example, "Return all events whose 50% confidence region intersects a circle centered on (α, δ) of radius r arcminutes." In addition to this search facility, it would also be useful to see events (and sets of events) displayed on a sky map:

Requirement 3.2.2. Individual GraceDB events, as well as groups of events (such as catalogs and query results), should be viewable on a map of the sky.

This way, search results could be displayed graphically with overlays such as galaxy catalogs, if desired. For example, an astronomer could search for all LowMass events in a specific time window with FAR $< 10^{-6}$ Hz. One way of satisfying this requirement would be by linking to the skymapViewer web service [8]. The details of this linkage are still being worked out.

Lastly, we note that the EMBB data models and API should be developed such that an *Observing Calculator* can be built against it. This would be a client tool such that, given a particular event and observing facility, the event's skymap would be used to produce a list of pointings. As with other derived analysis products, the logic for producing this list should be external to GraceDB. The final product (*i.e.*, a list of pointings or observing plan) could be archived by GraceDB, however. The exact specifications and details in the implementation of an *Observing Calculator* will be defined following input and discussion with the astronomical community participating in the LIGO-Virgo EM followup program.

3.3 Federated authentication and authorization

For the purposes of coordinating with the LIGO Open Science Center (LOSC) and EM observing partners, GraceDB must accept authenticated users who do not have LIGO.org credentials. Thus,

Requirement 3.3.1. GraceDB must support federated identities for authentication.

At the very least, it will be necessary to federate with the LIGO Guest IdP, as guest identities could be issued to external users. It would be even better, however, to federate with a larger group of institutions (such as InCommon [9] members) in order to avoid issuing large numbers of guest identities.

In addition, it will be necessary to restrict access to certain data within GraceDB to particular groups of users. For example, a member of the interested public should *not* see events that have not yet been reviewed and approved for release by the relevant authorities. The most straightforward way of accomplishing this would be to define groups of users and to set permissions based on group membership.

Requirement 3.3.2. GraceDB must be able to protect individual events with group-based authorizations. Permissions to create, view, and annotate events should be separately specifiable.

It will likely also be necessary to protect individual annotations in the same way. Naturally, in order to perform the necessary authorization checks, GraceDB will need group information

for all non-anonymous users. The Shibboleth project provides a convenient way of getting information about a user's groups into the GraceDB auth middleware. It would thus be desirable if the necessary group information were conveyed in the Shibboleth assertion provided by the user's IdP (or by an *Attribute Authority* set up specifically for this purpose). Handling all authentication/authorization with Shibboleth would greatly streamline the GraceDB auth infrastructure. It would, for example, obviate the need to cache information about user certificates from the LIGO LDAP.

Some aspects of this proposed auth infrastructure impose requirements external to GraceDB development. For example, a registration workflow will be needed for those without LIGO.org identities. Registration should result in their group membership information being provisioned into an attribute provider or LDAP. Group membership information will also be needed for robotic users.

3.4 Event approval workflow support

A candidate GW event must pass several checks before it is sent to astronomy MOU partners via GCN/TAN. Min-A Cho and Peter Shawhan have developed a preliminary version of the workflow for approval (see [10]), which will engage automatic followup checks and also (potentially) human-in-the-loop checks. The GraceDB event and annotation models will need to be modified slightly in order to support this approval workflow. The logical processing of the workflow itself, however, will be delegated to an automated followup coordinator (such as `gdb_processor`).

On the GraceDB server side, the event model will likely be extended to include fields for the event's iDQ (data quality) check status (passed, unknown, failed) and a sci-mon signoff for each detector (sci-mon user name, status: pass/fail, and optional comment). Changes in an event's iDQ status or sci-mon status should also generate LVAlerts in order to notify `gdb_processor` that the next steps in approval process may be taken. Lastly, the new labels SKYMAP_READY and PE_READY (in addition to the existing EM_READY label) would need to be introduced to the database.

Additional event model fields and/or labels could be added in the future to indicate, for example, an event's review and release status. Development of such features will likely proceed on an as-needed basis in response to scoped tests of the infrastructure.

3.5 Audience-specific rendering templates

GraceDB (in association with LOSC) should eventually provide information about candidate GW events to scientists outside of the LVC and to interested members of the general public. Thus, there are (at least) three audiences for GraceDB:

- LVC members
- EM astronomers in partnership with the LVC
- Members of the general public

Requirement 3.5.1. The presentation of information in GraceDB should be tailored to the appropriate audience. Thus, multiple audience-specific rendering templates will be required.

For example, an EM astronomer looking at an event as a potential candidate for follow-up observations may not be interested in details such as which analysis pipeline detected the

event. In contrast, she/he would likely be very interested measures of the event's significance. For each audience, the presentation should foreground the most important information.

The controller methods in GraceDB would choose a specific rendering template based on a user's group memberships (see Section 3.3). If a finer-grained customization is eventually desired, infrastructure could be introduced for users to create individual profiles to control presentation.

3.6 Service redundancy

It is very important that the GraceDB server be robust, since the scientific activities of the LVC and EM followup community could be negatively impacted by service degradations. At present, no special measures have been taken to build redundancy into the GraceDB deployment (beyond nightly, off-site backups). One one occasion during ER5, a sudden increase in the number of requests to GraceDB slowed the server to the point of unresponsiveness. Though this situation is not representative of normal operating conditions, it would be best to be prepared for such events in the future.

> **Requirement 3.6.1.** In the event that the GraceDB server becomes unresponsive, another instance should automatically take its place ("hot failover").

To this end, we have acquired a load balancing appliance at UWM in order to achieve redundancy with hot failover. The details of the configuration are still under discussion, but one possible setup would be to have three GraceDB instances: one that supports reading and writing, a second that is read-only, and a third read/write instance for use as a hot spare. This imposes a few additional requirements:

> **Requirement 3.6.2.** The load balancing appliance should support node selection by HTTP verb so that reading and writing operations are directed to the correct node.

This requirement also implies that the load balancer will have to do SSL off-loading, since the request will need to be decrypted in order to determine whether reading or writing is intended. Also,

> **Requirement 3.6.3.** The GraceDB database must be replicated across nodes, and a common filesystem for event files must be available to all nodes.

Lastly, since users are likely to make both read and write requests in the same session, and these requests will be directed to different nodes,

> **Requirement 3.6.4.** User sessions should be replicated across all nodes.

4 Future directions

In the process of working toward the requirements for the advanced detector era (Section 3), many other avenues of development will undoubtedly be discovered. For example, performance tuning is likely to assume a larger share of the development effort, particularly after all of the basic infrastructure components have been added. In fact, an in-depth performance analysis of GraceDB has yet to be performed. Leveraging JavaScript datastores for the event presentation may be one way of improving performance, since more of the data manipulation tasks can be moved to the client side. In addition, the design of the presentation and the handling of various media types (from audio to skymaps) will also be important considerations going forward.

A REST client usage

The Python module `ligo.gracedb.rest`, which is distributed with LALSuite, may be used "as is" to submit and modify events, or may be taken as a starting point for developing a custom interface. For example, the lines below demonstrate how to create a new CBC LowMass event and then replace it later:

```
import os, json
from ligo.gracedb.rest import GraceDb

# Build path to event file
eventFile = os.path.join(os.getcwd(),"coinc.xml")

# Instantiate the client.
gracedb = GraceDb()

# Create the event.
r = gracedb.createEvent("CBC","LowMass", eventFile).json()
graceid = r["graceid"]

# ... suppose some process modifies coinc.xml ...

# Replace the event with the new coinc file.
r = gracedb.replaceEvent(graceid, "coinc.xml")
```

Note that, before this client can be utilized, a valid proxy certificate must be generated using `ligo-proxy-init`. (For robotic users, simply set the environment variables X509_USER_CERT and X509_USER_KEY to point to the appropriate files.)

The GraceDb client class also includes the following additional methods:

- `events()` for accessing a list of events,
- `files()` for downloading a file or list of files, and `writeFile()` for uploading,
- `logs()` for obtaining a list of log entries, and `writeLog()` to create a new one,
- `labels()`, `writeLabel()`, and `removeLabel()` for managing labels,
- `tags()`, `createTag()`, and `deleteTag()` for managing tags.

The methods listed above expose most of GraceDB's functionality. Below is an example in which an event's VOEvent representation is obtained form GraceDB, written to a file, and then submitted to GCN via Comet [11]. (Code similar to this snippet was used to test GCN/TAN connection during ER5.)

```
import subprocess
from ligo.gracedb.rest import GraceDb

# Somehow decide which event to get
graceid = 'GXXXXXX'

# Instantiate the client.
gracedb = GraceDb()
```

```
# Pull down the VOEvent and write to tmp file.
url_template = gracedb.templates['event-vo-detail-template']
r = gracedb.get(url_template.format(graceid=graceid))
voevent = r.json()
tmpfile = open('/tmp/voevent.tmp',"w")
tmpfile.write(voevent)
tmpfile.close()

# Send it out with comet!
cmd = "comet-sendvo -p XXXX -f /tmp/voevent.tmp"
proc = subprocess.Popen(cmd, shell=True, stdout=subprocess.PIPE,
    stderr=subprocess.PIPE)
output, error = proc.communicate(voevent)
```

Finally, here is an example that retrieves events according to a complex query, creates a plot for each event, and uploads each plot with an accompanying log message:

```
from ligo.gracedb.rest import GraceDb, HTTPError

# Define the query for the ER4 catalog. The ER4 token in the
# search string limits the gpstimes to the ER4 epoch.
query = 'ER4 -INJ & EM_READY FAR<1e-12'

# Instantiate the client.
gracedb = GraceDb(SERVICE)

# Get the catalog events.
events = gracedb.events(query)

# Process events:
for event in events:
    graceid = event['graceid']
    gpstime = int(event['gpstime'])

    # ... invoke code to create a nifty plot

    tagname = 'sky_loc'
    message = 'My interesting sky localization plot.'
    filename = 'myskyplot_%s.png' % graceid

    try:
        r = gracedb.writeLog(graceid,message,
            filename=filename,tagname=tagname)
    except HTTPError:
        print "Error! Status code = %d", r.status
```

We emphasize that the REST API is not bound to be used with Python. Users may code against the API in any language that supports the necessary HTTP and X509 functionality. For example, `curl` can be used to create a new event as follows:

```
curl -X POST -F  "group=CBC" -F "type=LM" -F "eventFile=@coinc.xml" \
```

```
--cert /tmp/x509up_u${UID} --key /tmp/x509up_u${UID} --insecure \
https://gracedb.ligo.org/api/events/
```

B Command-line client usage

A GraceDB command-line client is also provided with LALSuite. This client simply wraps the functionality of the REST client described in the previous section. A detailed description of the usage of the command-line client may be obtained by typing:

```
gracedb --help
```

For example, to create a new CBC LowMass event using the file "coinc.xml," simply enter:

```
gracedb CBC LowMass coinc.xml
```

The command-line client can also be used to ping the server, search for events, upload and download files, create log entries, add labels, tag existing log entries, delete tags, and replace events. These functions are described in detail in the help message. By default, search results are presented in a table of tab-separated values, but LigoLw is also available for events which have "coinc.xml" files. One tricky aspect of the command-line client is that certain characters (such as quotation marks) need to be escaped. For example, to search for events submitted during ER5 by the GSTLAL SPIIR analysis robot:

```
gracedb search ER5 submitter: \"gstlal-spiir\"
```

References

[1] http://djangoproject.com

[2] L. Richardson and S. Ruby, *RESTful Web Services* (O'Reilly, Sebastopol, CA, 2007).

[3] http://www.django-rest-framework.org

[4] https://bugs.ligo.org/redmine/issues/1367

[5] http://httpd.apache.org

[6] https://shibboleth.net

[7] http://www.ivoa.net/documents/latest/UCDlist.html

[8] https://www.ligo.caltech.edu/ rwilliam/skymapViewer

[9] http://www.incommon.org

[10] M. Cho, https://dcc.ligo.org/LIGO-T1400414 (2014).

[11] https://github.com/jdswinbank/Comet