Notes on VOEvent Documentation

Release 0

John Swinbank & Tim Staley

November 02, 2016

Contents

VOEvent is the International Virtual Observatory Alliance (IVOA) recommended mechanism for describing astronomical transients. Specifically, the VOEvent standard *(Seaman, 2011)*

> defines the content and meaning of a standard information packet for representing, transmitting, publishing and archiving information about a transient celestial event, with the implication that timely follow-up is of interest.

This document provides an introduction to working with VOEvents. It is not normative; rather, it represents the authors' own particular biases and experiences. This material is intended to accompany the "VOEvent Hands-On Session" presented at Hot-Wiring the Transient Universe IV (henceforth "Hotwired 4") in May 2015; we hope, though, that it will prove generally useful.

We assume a working knowledge of Python.

CHAPTER 1

What is a VOEvent and why would I want one?

1.1 The transient deluge

Systems for detecting astrophysical transients are becoming increasingly ubiquitous and high volume: from space-based gamma-ray monitors like NASA's Fermi and Swift missions, through present-day ground based surveys in both the optical (e.g. the Catalina Real-Time Transients Survey and the Palomar Transients Factory) and radio (the LOFAR Radio Sky Monitor and ASKAP's Variables and Slow Transients survey), to future facilities such as the Square Kilometre Array (SKA) and Large Synoptic Survey Telescope (LSST). LSST alone promises to detect and announce up to 40 million transients per night when its survey begins early in the 2020s.

Obtaining the best scientific results from these facilities requires timely and accurate follow-up observations of the most relevant detections. However, it is obviously humanly impossible—even given a *large* cohort of graduate students—to read and understand millions of transient alerts per night, let alone to select those which merit further observation. Even were it possible, humans are slow: slow at reading, but, even more so, slow at negotiating with their peers to organize telescope time.

1.2 The VOEvent standard

It is this problem which VOEvent seeks to alleviate. It provides a standardized, machine-readable means of describing transient celestial events. It enables the author to specify:

- *Who* they are;
- *What* they have observed;
- *Where and when* the observed it;
- *How* the observations were made;
- *Why* they think it is of general interest to the community.

In addition, it is possible for them to provide citations to previously announced events, and to refer to other content outside the context of the VOEvent itself. In short, the aim is to provide the reader not just with a notification that something was observed, but to encapsulate as much information about it as is practical, so that they can make an informed decision about whether to perform a follow-up observation.

Let's emphasize that last sentence again: the *recipient* of a VOEvent makes a decision about whether to perform any action in response to a VOEvent. The event itself does not constitute a trigger or an instruction to the recipient, and it does not suggest what sort of follow-up might be appropriate. It describes only the event as seen by the observer.

VOEvents are XML documents. XML is a plain text "markup" language: it makes it possible to annotate a document to show its structure in a way that a computer can understand, while also being human readable (for a broad definition of "readable" and a fairly tolerant human). The VOEvent standard defines an XML "schema", which describes the subset of all possible constructs which it is legal to use in a VOEvent document. As we'll see *later*, it's perfectly possible to read and write VOEvents using your favourite text editor. However, that's a lot of effort; there is a range of tools and libraries you can deploy to make your life easier. We'll *cover them too*.

The VOEvent standard was defined by *Seaman et al. (2011)* and it has an official stamp of approval from the IVOA. It is currently curated by the IVOA's Time Domain Interest Group, or TDIG.

1.3 Distributing VOEvents

It's important to realise that the VOEvent standard says nothing of any significance about what you should do with a VOEvent once you've got one. In particular, it might seem reasonable to assume that once the discoverer of a transient has encapsulated it in a VOEvent document, they might wish to distribute it to the community, but the mechanisms by which they can do that are *not* defined by the standard. It's perfectly to do this by any convenient means: e-mail, post, semaphore, ...

This is both a blessing and a curse. It's good architecture to separate the representation of events from their transmission, and it means the standard can be relatively concise and manageable. Plus there's a great deal of flexibility here: if a project adopts VOEvent, it doesn't tie them into some particular networking infrastructure. However, for somebody who just wants to find news on the latest transients, or who just wants to ensure that their events are available to the community, there's no obvious place to start.

This situation is partially mitigated by the VOEvent Transport Protocol (VTP; *Allan & Denny, 2009*). This is an IVOA Note, rather than a Recommendation; in other words, it is a suggestion which has not been through the IVOA's rigorous review process. There is a revised version of this protocol in preparation, with the hope of developing it into a fully-fledged IVOA standard in future. The changes are intentionally being kept minimal, and focus more on resolving ambiguities in the specification than on changing the on-the-wire protocol.

VTP is minimal by design: it builds upon low-level networking primitives to create something that gets the job done. Arguably, a more modern system would be layered on top of higher level standards, like HTTP, which would provide a more robust basis and make it easier to take advantage of existing infrastructure. However, VTP is the nearest we have to a standard at the moment, and it certainly provides mechanisms for sending and receiving streams of VOEvents with relatively low latency. This document *will describe* how to connect up to and participate in the existing VTP network.

1.4 Higher level functionality

We conclude this section with a warning. All of the tools described here are fairly low-level: they provide mechanisms for reading, writing and exchanging individual VOEvent packets. This is necessary but not sufficient to service future science projects. We need to build upon these lower level tools to construct event aggregators and filters and classifiers and distribution systems which make it possible for the astronomer to reason about and respond to VOEvents in bulk. While some important work has been done on this (e.g. *Williams et al, 2009*, *Poci et al, 2015*), there's still a long way to go. Maybe after reading this you will be inspired to contribute.

We'll return to this point later in considering *some of the challenges* facing VOEvent in the future.

CHAPTER 2

Finding and installing useful tools

We'll start by installing two useful tools for working with VOEvents in the Python programming language. It's impossible to provide comprehensive instructions for every possible operating system, but the good news is that the tools are generally pretty easy to install. We'll discuss then general principles, and provide step-by-step instructions for common platforms—you should be able to adapt these to your own system.

Broadly speaking, there are two possible approaches you can take to getting an appropriate set of tools on your system: you can either build & install them yourself—the "conventional" way—or use some sort of prepackaged environment which has everything you need already installed in it. This document focuses on the first option, but includes a *brief discussion* of the second approach.

2.1 Background and dependencies

In general, we suggest relying on package management tools where possible, rather than attempting to compile everything from scratch. For Python packages, we'll use pip; for system libraries, use a package manager appropriate to your operating system (APT for Debian, Ubuntu and related systems; Macports or Homebrew on OS X, etc). If you don't have permission to use one of these systems, ask your system administrator nicely (or, better, get yourself a cheap-as-chips VM from Amazon EC2 or Digital Ocean or similar).

Note that we depend on Python 2.7; not all of the tools described have yet been ported to Python 3. This is unfortunate, but you can help.

We suggest using Virtualenv to create an isolated environment for installing the Python packages we'll need. That means you don't need any special privileges, and you can blow the whole thing away and start over when you make a mistake simply by running `rm`.

In addition to Python, pip and Virtualenv, you will need to install libxml2 and libxslt. You'll need the header files for zlib (the library itself was almost certainly installed along with your operating system). You'll also find IPython useful for following along with our examples. This is a good time to look into package managers, as mentioned above. On Ubuntu 14.04.2:

```
$ sudo apt-get update
$ sudo apt-get install libxml2-dev libxslt-dev zlib1g-dev python-dev python-pip python-virtualenv ipy
```

Or with OS X using Macports, try:

```
$ sudo port selfupdate
$ sudo port install libxml2 libxslt python27 py27-pip py27-virtualenv py27-ipython

# Use the software we've just installed as the default versions
$ sudo port select python python27
$ sudo port select pip pip27
$ sudo port select virtualenv virtualenv27
$ sudo port select ipython ipython27
```

Similar incantations should work on other systems. Next, create a Virtualenv for working in and activate it. The name is arbitrary; call it whatever you like:

```
$ virtualenv voevent-venv
New python executable in voevent-venv/bin/python
Installing setuptools, pip...done.
$ . voevent-venv/bin/activate
```

You will need to source the `activate` script in this way every time you want to use this environment.

2.2 Reading and writing VOEvents: voevent-parse

For manipulating the contents of VOEvents, we'll use the voevent-parse library. After you have set up and activated your Virtualenv, as above, installing voevent-parse is easy using pip:

```
$ pip install voevent-parse
```

The compilation will take a few moments. Then you can check if it is properly installed by running:

```
$ ipython
[...]
In [1]: import voeventparse

In [2]: voeventparse.__version__
Out[2]: 0.7.0
```

If you aren't familiar with the IPython shell, it's worth your while to spent a few minutes getting used to it at this point.

This document will provide an introduction to using voevent-parse, but for the full story you should refer to its fine documentation.

2.3 Sending and receiving VOEvents: Comet

For transmitting VOEvents using VTP, we'll use Comet (*Swinbank, 2014*). Installation proceeds in just the same way as for voevent-parse:

```
$ pip install comet
```

Unlike voevent-parse, Comet is not a library you will import into Python, but is rather invoked through `twistd`, a stand alone tool. You can check that it is installed by running:

```
$ twistd comet --help
Usage: twistd [options] comet [options]
Options:
  -r, --receive                  Listen for TCP connections from authors.
  -b, --broadcast                Re-broadcast VOEvents received.
[...]
```

It's also worth running the Comet test suite to check that everything has been installed properly:

```
$ trial comet
comet.handler.test.test_relay
  EventRelayTestCase
      test_interface ...                  [OK]
          test_name ...                   [OK]
[...]
Ran 144 tests in 0.443s

PASSED (successes=144)
```

All tests should succeed.

Again, this guide will cover the basics of using Comet, but it is well worth your while to read its documentation.

2.4 An alternative approach: containerization

Note: This section was contributed by Casey Law.

Finally, you can use docker to quickly build all of the above for use in a Jupyter (IPython) notebook. Docker uses "containerization" (a kind of virtual machine) to build entire operating systems and save you a lot of work. If you choose to go this route, you'll need to install docker. Then download the Dockerfile for the docker notebook to an empty directory and run:

```
$ docker build -t notebook .
$ docker run -d -p 8888:8888 -v ~/jupyter-notebooks/:/ipynb notebook
```

Note that `~/jupyter-notebooks` is a stand-in for the path to a place on your file system where you'd like to save your notebooks. The `docker run` command start the Jupyter notebook server running the entire VOEvent environment. You can find the notebook by pointing your browser at http://192.168.59.103:8888.

CHAPTER 3

A look inside a VOEvent

As we've discussed, VOEvents are designed to be readable both by machines and by humans. Let's start by looking at a real live VOEvent by eye and picking out the important characteristics. Here's one:

```
<?xml version="1.0" encoding="UTF-8"?>
<voe:VOEvent xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
             xmlns:voe="http://www.ivoa.net/xml/VOEvent/v2.0"
             xsi:schemaLocation="http://www.ivoa.net/xml/VOEvent/v2.0
                                 http://www.ivoa.net/xml/VOEvent/VOEvent-v2.0.xsd"
             version="2.0" role="test"
             ivorn="ivo://org.hotwired/exciting_events#123">
  <Who>
    <AuthorIVORN>ivo://hotwired.org</AuthorIVORN>
    <Date>1970-01-01T00:00:00</Date>
    <Author>
      <title>Hotwired VOEvent Tutorial</title>
    </Author>
  </Who>
  <What>
    <Group name="source_flux">
      <Param dataType="float" name="peak_flux" ucd="em.radio.100-200MHz"
             unit="Janskys" value="0.0015">
        <Description>Peak Flux</Description>
      </Param>
      <Param dataType="float" name="int_flux" ucd="em.radio.100-200MHz"
             unit="Janskys" value="2.0e-3">
        <Description>Integrated Flux</Description>
      </Param>
    </Group>
  </What>
  <WhereWhen>
    <ObsDataLocation>
      <ObservatoryLocation id="GEOSURFACE"/>
      <ObservationLocation>
        <AstroCoordSystem id="UTC-FK5-GEO"/>
        <AstroCoords coord_system_id="UTC-FK5-GEO">
          <Time unit="s">
            <TimeInstant>
              <ISOTime>1970-01-01T00:00:00</ISOTime>
            </TimeInstant>
          </Time>
          <Position2D unit="deg">
            <Name1>RA</Name1>
            <Name2>Dec</Name2>
```

```
41             <Value2>
42               <C1>123.5</C1>
43               <C2>45</C2>
44             </Value2>
45             <Error2Radius>0.1</Error2Radius>
46           </Position2D>
47         </AstroCoords>
48       </ObservationLocation>
49     </ObsDataLocation>
50   </WhereWhen>
51   <How>
52     <Description>JDS spied with his little eye</Description>
53   </How>
54   <Why importance="0.5">
55     <Inference probability="0.1" relation="identified">
56       <Name>GRB121212A</Name>
57       <Concept>process.variation.burst;em.radio</Concept>
58     </Inference>
59   </Why>
60   <Citations>
61     <EventIVORN cite="followup">ivo://org.hotwired/exciting_events#1</EventIVORN>
62   </Citations>
63 </voe:VOEvent>
```

The first thing to realise is that it's *not* important to understand all the details. There's a lot of "boilerplate" markup language which isn't necessary to getting a good overview of what's going on here. However, it should be pretty easy to tease out a few salient points.

3.1 Overall structure

The first thing that should be immediately obvious are the blocks referring to the who, what, where & when, how, and why that we *previously discussed*.

The `Who` block in lines 8 to 14 tells us that this event was generated back in 1970 by an author who was eagerly anticipating this tutorial session. `What` that author observed is described by lines 15 through 26 it was a radio source, for which we provide peak and integrated flux densities in Janskys at a frequency between 100 and 200 MHz. `WhereWhen` (lines 27–50) tells us that the observation was made at an observatory on the Earth's surface, and provides an RA and dec with association uncertainties. `How` just provides a free-form text description of how the observation was made, and `Why` identifies this with a GRB taking place in 2012 (nearly 43 years after this observation was made—not bad!).

3.2 Event roles

While, in general, we're going to avoid considering the subtleties of the markup, there are a few features it's worth pointing out. First, right up the top of the event we have the string:

```
role="test"
```

This is *important*. It serves as notice to recipients that the event does not describe an actual astronomical event. If you are experimenting with VOEvent, you should *always* use the `test` role, even if you don't intend for your events to be published: if one should leak of your sandbox, it could be horribly expensive if somebody mistakes it for a notice of a genuine transient and triggers their multi-hundred-million-dollar followup facility.

That said, there are three other possible roles we should mention. `observation` is the obvious one: it's reporting the results of a particular observation. `utility` is informing the recipient about details of the observing system; for example, a change in configuration. Finally, `prediction` indicates that the author is indulging in a little fortune telling.

3.3 IVORNs and identifiers

Next, let's talk about IVORNs, or "IVOA Resource Names". IVORNs provide unique identifiers for entities known to the virtual observatory, where "entity" is quite broadly defined. In particular, here we provide an identifier both for the author of the event:

```
<AuthorIVORN>ivo://org.hotwired</AuthorIVORN>
```

and for the event itself:

```
ivo://org.hotwired/exciting_events#123
```

First, consider the identity of the author: `org.hotwired`. In formal IVOA terms, this should be a "naming authority" which has the right to create IVOA-compliant identifers for the resources it creates. In practice, the necessary registration material to make this worthwhile for VOEvents isn't yet in place, so we'll improvise: choose a string which you are confident will usefully identify you (or your facility) to the recipient, and which is unlikely to clash with anybody else issuing VOEvents. Using a straightforward internet host (DNS) name (like `hotwired.org`) is discouraged (folks might try to resolve it through the normal internet systems, which will fail), but reversing it to create something which is uniquely tied to a given organization but which can't be confused with a regular host name should work, and that's exactly what we've done here.

Now, look at the identity of the event itself. This is particularly worth noting, as the VOEvent standard assigns specific meanings to each part. In this case, we are considering event number 123 published to the `exciting_events` stream by `org.hotwired`. In general, we expect events to be grouped into streams in this way, where a stream represents a specific source of events—a given instrument or a particular way of processing the data, say. A single naming authority could manage multiple streams (so we could imagine `org.hotwired` also publishing to the `tedious_events` stream).

We can refer to other events by specifying their IVORN. For example, this event specifies:

```
<Citations>
  <EventIVORN cite="followup">ivo://org.hotwired/exciting_events#1</EventIVORN>
</Citations>
```

This means that this VOEvent is reporting a follow-up to a previous observation, which was described by event 1 in the same stream. As well as `followup`, it is also possible for the current event to use `supersedes` to provide a corrected version of a previous event, or `retraction` to withdraw it altogether. Ultimately, an event aggregation service could group together VOEvents which cite each other to build up a full description of a particular astronomical event.

At time of writing, there is no central body for VOEvents which creates naming authorities or enforces standards on IVORNs. For now, you're simply encouraged to structure your IVORNS following the above guidelines. Longer term, work is ongoing to integrate the VOEvent system with the wider VOEvent "registry" system, which will help ensure standards are enforced and provide mechanisms to look up details of particular authors, available event streams, and the events themselves.

3.4 Unified Content Descriptors

Depending on the source of the event, it might be “obvious” to the human that “peak flux” measured in Janskys likely refers to some sort of electromagnetic radiation. However, the computer which might be automatically processing and making decisions based on this VOEvent needs all the help it can get. To that end, we can annotate the event with Unified Content Descriptors, or UCDs (*Derriere et al, 2005*). For example, we write

```
ucd="em.radio.100-200MHz"
```

to indicate the type of measurement that `peak_flux` and `int_flux` refer to. A very similar idea applies to the scientific inference:

```
<Concept>process.variation.burst;em.radio</Concept>
```

CHAPTER 4

Manipulating VOEvents in code

We've *seen* that VOEvents are just text documents, and you can happily read and write them in your favourite text editor. That's fine as far as it goes, but it's not a practical approach for building a large scale event handling system. Luckily, one of the advantages of using XML to encode VOEvents is that we can tap into a wide range of infrastructure and tools which have been developed specifically for processing XML. Effectively every mainstream programming language has a whole suite of libraries for handling XML available—see, for example, the documentation for the Python standard library.

While it's possible to directly apply these generic XML handling tools to VOEvents, it's even more convenient to use a library which has been especially developed to work with VOEvents. Here, we are using voevent-parse which does just that. Voevent-parse builds on the popular lxml library.

The examples below should be just enough to get you started with voevent-parse. An expanded version which goes into a little more detail on the syntax quirks of the lxml library is also available in IPython Notebook format as the voevent-parse-tutorial, In addition, we refer you to lxml's documentation for full details.

4.1 Parsing a VOEvent

Let's start by reading some basic data from a VOEvent. We'll save the example we looked at *earlier* to a file—say, `voevent.xml`—and read that into Python. You can follow along using IPython.

First, we import the `voeventparse` library into Python, and use it to load the `voevent.xml` file from disk:

```
In [1]: import voeventparse as vp

In [2]: with open('voevent.xml') as f:
   ...:     v = vp.load(f)
   ...:

In [3]: v
Out[3]: <Element VOEvent at 0x104747560>
```

The basic "attributes" (the `role` and the `ivorn`, as well as some of the XML boilerplate that we skipped over) of the root `VOEvent` element are accessible as a dictionary on `v`:

```
In [4]: v.attrib['ivorn']
Out[4]: 'ivo://org.hotwired/exciting_events#123'

In [5]: v.attrib['role']
Out[5]: 'test'
```

We can also descend into the sub-elements of the event and retrieve the key information:

```
In [6]: v.Who.Author.title
Out[6]: 'Hotwired VOEvent Tutorial'

In [7]: v.What.Group.Param.Description
Out[7]: 'Peak Flux'

In [8]: v.What.Group.Param.attrib['value']
Out[8]: '0.0015'
```

Wait a minute—there are actually two `Param` elements inside `What`. How do we get the integrated flux? Two possibilities, actually. First, you can iterate over the children of the parent element:

```
In [9]: for element in v.What.Group.iterchildren():
   ...:     print element.attrib['name'], ": ", element.attrib['value']
   ...:
peak_flux : 0.0015
int_flux : 2.0e-3
```

Alternatively, you can search for particular elements:

```
In [10]: v.find(".//Param[@name='int_flux']").attrib['value']
Out[10]: '2.0e-3'
```

It's worth noting that the VOEvent schema does *not* guarantee that a VOEvent will provide a `Param` with a name of `int_flux`. The flexibility of VOEvent is both a blessing and a curse here: since different sources of events will likely report on quite different types of observation, it's hard to make sense of them without already having some idea of what you expect to find inside. The good news is that you'll probably have quite a good idea of which event *streams* you're interested in, and you can expect (although it's not guaranteed...) events with in a particular stream to follow a regular structure.

On the subject of the schema: you'll come across all sorts of weird constructs that people have stashed in VOEvents, and you might even be tempted to come up with some yourself. Often, folks will be able to figure out what you mean, and XML handling tools generally try to be quite liberal in what they'll accept. However, the schema exists for a reason: if you stray outside its boundaries, all bets are off regarding whether your events will be interpreted in the way you intend. Indeed, many systems will simply refuse to accept an event which isn't valid according to the schema. Conveniently, voevent-parse can check this for you:

```
In [11]: vp.valid_as_v2_0(v)
Out[11]: True
```

Note that we are checking against the VOEvent version 2.0 schema, which is the latest version and is definitely what you should be using.

4.2 Creating a VOEvent

Having seen how we can use voevent-parse to read a VOEvent, the next step is to create one. In order to make this a bit more interesting, let's use a real transient. Gaia has recently started publishing Photometric Science Alerts to a publicly-accessible website, but they aren't yet available as VOEvents. Let's pick on the following:

Name	Observed	Published	RA (deg.)	Dec. (deg.)	Magnitude	Historic mag.	Historic scatter	Class	Comment
Gaia14adi	2014-11-07 01:05:09	2014-12-02 13:55:54	168.47841	-23.01221	18.77	19.62	0.07	unknown	Fading source on top of 2MASS Galaxy (offset from bulge)

We'll start by creating the skeleton of our VOEvent packet. We carefully to set the role to `test` so that nobody is tempted to start acting on the contents of this demo event. We also set the timestamp in the `Who` block to the time the event was generated (*not* when the observation was made), as per the specification:

```
In [1]: import voeventparse as vp

In [2]: import datetime

In [3]: v = vp.Voevent(stream='hotwired.org/gaia_demo', stream_id=1,
                       role=vp.definitions.roles.test)

In [4]: vp.set_who(v, datetime.datetime.utcnow())
```

Now to define the author. Note that this is *us*, since we're generating the VOEvent—this isn't an official Gaia product, and we neither want to claim credit for the result ourselves, nor do we want people to start hassling the Gaia folks with questions about our event. We'll make sure that's noted in the explanatory text attached to the event:

```
In [5]: vp.set_author(v, title="Hotwired VOEvent Hands-on",
                      contactName="John Swinbank")

In [6]: v.Description = "This is not an official Gaia data product."
```

Now let's add details of the observation itself. We'll record both the magnitude that Gaia is reporting for this particular event, and the historic values they also provide:

```
In [7]: v.What.append(vp.Param(name="mag", value=18.77, ucd="phot.mag"))

In [8]: h_m = vp.Param(name="hist_mag", value=19.62, ucd="phot.mag")

In [9]: h_s = vp.Param(name="hist_scatter", value=0.07, ucd="phot.mag")

In [10]: v.What.append(vp.Group(params=[h_m, h_s], name="historic"))
```

Now we need to specify where and when the observation was made. Rather than trying to specify a position for Gaia, we'll just call it out by name. Note that Gaia don't provide errors on the position they cite, so we're rather optimistically using `0`:

```
In [11]: vp.add_where_when(v,
                           coords=vp.Position2D(ra=168.47841, dec=-23.01221, err=0, units='deg',
                                                system=vp.definitions.sky_coord_system.fk5),
                           obs_time=datetime.datetime(2014, 11, 7, 1, 5, 9),
                           observatory_location="Gaia")
```

We should also describe how this transient was detected, and refer to the name that Gaia have assigned it. Note that we can provide multiple descriptions (and/or references) here:

```
In [12]: vp.add_how(v, descriptions=['Scraped from the Gaia website',
                                     'This is Gaia14adi'],
                    references=vp.Reference("http://gsaweb.ast.cam.ac.uk/alerts/"))
```

Finally, we can provide some information about why this even might be scientifically interesting. Gaia haven't provided a classification, but we can at least incorporate the textual description:

```
In [13]: vp.add_why(v)

In [14]: v.Why.Description = "Fading source on top of 2MASS Galaxy (offset from bulge)"
```

Finally—and importantly, as we discussed above—let's make sure that this event is really valid according to our schema:

```
In [15]: vp.valid_as_v2_0(v)
True
```

Great! We can now save it to disk:

```
In [16]: with open('gaia.xml', 'w') as f:
             vp.dump(v, f)
```

And we're all done. You can open the file in your favourite editor to see what we've produced, but note that it probably won't be particularly elegantly formatted. You can use a tool like `xmllint` to pretty print it; you should end up with something like:

```
<?xml version="1.0" encoding="UTF-8"?>
<voe:VOEvent xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:voe="http://www.ivoa.net/xm
 <Who>
   <Description>VOEvent created with voevent-parse: https://github.com/timstaley/voevent-parse</Desc
   <Date>2015-04-17T15:52:56</Date>
   <Author>
     <title>Hotwired VOEvent Hands-on</title>
     <contactName>John Swinbank</contactName>
   </Author>
 </Who>
 <What>
   <Param dataType="float" name="mag" ucd="phot.mag" value="18.77"/>
   <Group name="historic">
     <Param dataType="float" name="hist_mag" ucd="phot.mag" value="19.62"/>
     <Param dataType="float" name="hist_scatter" ucd="phot.mag" value="0.07"/>
   </Group>
 </What>
 <WhereWhen>
   <ObsDataLocation>
     <ObservatoryLocation id="Gaia"/>
     <ObservationLocation>
       <AstroCoordSystem id="UTC-FK5-GEO"/>
       <AstroCoords coord_system_id="UTC-FK5-GEO">
         <Time unit="s">
           <TimeInstant>
             <ISOTime>2014-11-07T01:05:09</ISOTime>
           </TimeInstant>
         </Time>
         <Position2D unit="deg">
           <Name1>RA</Name1>
           <Name2>Dec</Name2>
           <Value2>
             <C1>168.47841</C1>
             <C2>-23.01221</C2>
           </Value2>
           <Error2Radius>0</Error2Radius>
         </Position2D>
       </AstroCoords>
     </ObservationLocation>
   </ObsDataLocation>
 </WhereWhen>
 <Description>This is not an offical Gaia data product.</Description>
 <How>
   <Description>Scraped from the Gaia website</Description>
   <Description>This is Gaia14adi</Description>
   <Reference uri="http://gsaweb.ast.cam.ac.uk/alerts/"/>
 </How>
 <Why>
   <Description>Fading source on top of 2MASS Galaxy (offset from bulge)</Description>
 </Why>
</voe:VOEvent>
```

CHAPTER 5

Subscribing to VOEvent streams

We now have a fairly good grasp of what a VOEvent is and what we can do with it when we've got one. However, we've still not discussed how we can actually go about getting our hands on some events.

5.1 A VOEvent Transport Protocol primer

At time of writing, the easiest way to go about this is by using the VOEvent Transport Protocol, or VTP, that we *mentioned previously*. Before proceeding, let's take a few moments to discuss how VTP works. For a more detailed description, the reader is referred to *Swinbank (2014)*.

The VTP system facilitates transmission of events from their *author* to one or more *subscribers*. Since it's impractical to suggest that a single author contact each subscriber individually, and it's likely that subscribers want to receive events from more than one author, we introduce a *broker* as an intermediary. When a subscriber is interested in receiving events, it connects to the broker to register its interest, and then it waits. When the author has an event to distribute, it uploads it to the broker, which then redistributes it to all connected subscribers.

We can expand this system by simply adding another broker, and causing our brokers to subscribe to each others outputs. Given a well connected network of brokers of this form, it doesn't ultimately matter to which broker an author sends their event, or to which broker a subscriber connects: all events are ultimately distributed to all entities in the network. This is shown schematically in the figure below.

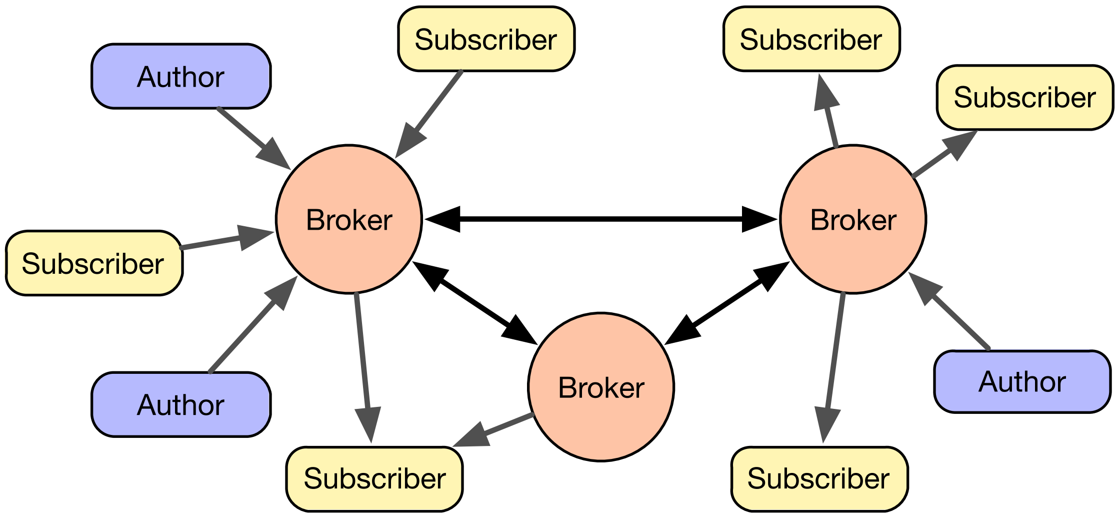

Fig. 5.1: An overview of a VTP network. Arrows indicate the direction of the data flow.

Such a network is robust: there is no single point of failure. If a particular broker fails, the authors and subscribers directly connected to it are cut off, but the rest of the network is not disrupted. Further, there's no downside to authors and subscribers connecting to multiple brokers simultaneously: the protocol is smart enough to eliminate duplicate events, so everybody only receives one copy of everything.

Of course, given the putative *transient deluge*, one copy of everything is likely far more than the subscriber is interested in (or, indeed, capable of ingesting). We'll discuss some simple filtering below, but this is certainly a problem for the *future*.

5.2 Subscribing to a broker

We'll start by using Comet (*Swinbank, 2014*) to subscribe to a feed of events from a broker. Assuming you've got Comet installed (see our *instructions*), you can control Comet using the `twistd` command. By default, `twistd` assumes you want to run Comet as a *daemon* (that is, a long running background process, which communicates with the outside world by log message rather than through your terminal). That's convenient a lot of the time, but it's something we want to avoid for the sake of this terminal. We can do so by giving `twistd` a `-n` flag, then telling it we want to run Comet:

```
$ twistd -n comet --help
Usage: twistd [options]
Options:
  -r, --receive                   Listen for TCP connections from authors.
  -b, --broadcast                 Re-broadcast VOEvents received.
[...]
```

As you can see from the help message, Comet has a lot of options. That's partly because it can act (simultaneously!) as both broker and subscriber on a VTP network. The good news is that we can skip over most of them for now: if you're curious, you can always refer to the Comet documentation for details.

There are just a couple of options we need to get started. The first is to decide on an identifier for our local system. Just like *VOEvents themselves*, entities on the network are identified by means of a URL-like `ivo://` identifier. It doesn't matter much what you choose, as long as you follow the required format. Let's go with `ivo://hotwired.org/test` for now.

Secondly, we need a broker to subscribe to. There are several listed at the TDIG site. For the purposes of this tutorial, we'll use `voevent.4pisky.org`, hosted by the 4 Pi Sky project.

Given that, we can start Comet running:

```
$ twistd -n comet --local-ivo=ivo://hotwired.org/test --remote=voevent.4pisky.org
2015-04-17 15:38:36-0400 [-] Log opened.
2015-04-17 15:38:36-0400 [-] twistd 13.2.0 (/opt/local/Library/Frameworks/Python.framework/Versions/2
2015-04-17 15:38:36-0400 [-] reactor class: twisted.internet.selectreactor.SelectReactor.
2015-04-17 15:38:36-0400 [INFO -] Subscribing to remote broker voevent.4pisky.org:8099
```

You'll see it print some information to the screen saying that it's starting up and subscribing to the broker... and then it will sit there, waiting to receive an event. Unfortunately, we can't guarantee when an event will be received: usually there's something happening every few minutes, but for now we'll just have to be patient. Then, finally:

```
2015-04-17 15:44:51-0400 [INFO VOEventSubscriber,client] VOEvent ivo://nasa.gsfc.gcn/Fermi#GBM_Test_F
```

5.3 Acting on events received

Of course, just getting a message to tell us that an event was received is only marginally thrilling. Even better is if we can actually see what that event contains. Conveniently, you can ask Comet to print a copy of events it receives using the `--print-event` option:

```
$ twistd -n comet --local-ivo=ivo://hotwired.org/test --remote=voevent.4pisky.org --print-event
[...]
2015-04-17 15:44:51-0400 [INFO VOEventSubscriber,client] VOEvent ivo://nasa.gsfc.gcn/Fermi#GBM_Test_P
2015-04-17 15:44:51-0400 [-] <voe:VOEvent xmlns:voe="http://www.ivoa.net/xml/VOEvent/v2.0" xmlns:xsi=
2015-04-17 15:44:51-0400 [-]   <Who>
2015-04-17 15:44:51-0400 [-]     <AuthorIVORN>ivo://nasa.gsfc.tan/gcn</AuthorIVORN>
2015-04-17 15:44:51-0400 [-]     <Author>
2015-04-17 15:44:51-0400 [-]       <shortName>Fermi (via VO-GCN)</shortName>
2015-04-17 15:44:51-0400 [-]       <contactName>Julie McEnery</contactName>
[...]
```

Those with real ambition might find even this underwhelming, so there exists a `--save-event` option which will dump the received events to files (by default in the current working directory; use the `--save-event-directory` option to tweak this).

Realistically, of course, you'll want to be able to take action when you receive an event. Comet gives you a couple of options here. One is to invoke a command whenever an event is received, passing it that event on standard input. You can use this to perform whatever logic you require. For example, we could make a quick & dirty VOEvent-to-e-mail gateway using a script like this:

```
#!/bin/bash

/usr/bin/mail -s "VOEvent Received" email@address.invalid <&0
```

Then simply:

```
$ twistd -n comet [...] --cmd=$(pwd)/mail.sh
```

Note that we have to provide the full path to the command to be executed, and be aware that this assumes you have an appropriate `/usr/bin/mail` set up on your local system.

This is obviously a pretty trivial example: the fourpiskytools package contains something rather more elaborate.

The really ambitious can go further still. Comet makes it possible to add "plug-in" code which is executed directly within Comet itself to handle events received. In fact, this is exactly how the `--print-event` and `-save-event` commands we used above are implemented. For more details, see Comet's documentation on Event Handlers, or refer to the the source of those commands for inspiration.

5.4 Filtering event streams

The VTP standard does not provide for a way to select which events you receive: brokers distribute all events received to all of their subscribers. That's OK as long as the volume of events remains relatively low and the subscribers are all willing to get their hands dirty writing scripts locally to extract the information they want. Ultimately, though, it's more efficient for subscribers to be able to select only those events they are interested in receiving from the broker.

Although this is not possible using vanilla VTP, Comet introduces its own extension to the protocol which enables the subscriber to ask the broker to send only events which match certain criteria to the subscriber. For this, we use the XPath XML query language. For example, to select only those events which were issued by VO-GCN (ie, originate from the NASA Gamma-ray Coordinates Network), we can use:

```
$ twistd -n comet [...] --filter="//Who/Author[shortName='VO-GCN']"
```

Of course, as usual when processing VOEvents, we need to know something about the structure of the events we're interested in. If we know, for example, that the event contains a `Sun_Distance` parameter, we can make selections based on that:

```
$ twistd -n comet [...] --filter="//Param[@name='Sun_Distance' and @value>40]"
```

Quite complex expressions are possible: for more examples, refer to the Comet documentation on filtering and *Swinbank (2014)*. Unfortunately, the XPath language is complex, and, as mentioned, you need to have a good understanding of the event stream you are trying to filter before you can construct useful queries. Furthermore, this is a Comet specific extension: it requires that *both* the subscriber *and* the broker be running Comet. If that's not the case, it won't do any harm to the network, but the filtering will not be applied.

However, there's a more far-reaching issue than the problems described above: although XPath is powerful, it doesn't ultimately provide the sort of filtering and selection capability that will be needed to handle the event volumes and complexities expected from future surveys. We'll *return* to this point. But first, let's discuss how to send events, as well as receive them.

Distributing VOEvents

We're now in a position not to just receive and act upon other VOEvents created by others, but to actually start giving back to the network by contributing our own events.

Warning: The tools described in this document make it possible for you broadcast events to all interested parties connected to the network. Please be considerate of their resources. In particular, tests should be *clearly* marked as such using the notation described *earlier*.

The good news is that Comet includes a tool that makes submitting your events to the network really straightforward. You can simply compose your event—using voevent-parse as *previously demonstrated* if you wish—and save it to a file on disk. Then use the `comet-sendvo` script included with Comet to submit it to a broker of your choice:

```
$ comet-sendvo -f voevent.xml
```

If you prefer, you can also feed your event to `comet-sendvo` on standard input:

```
$ <voevent.xml comet-sendvo
```

Easy as pie, right?

Well, sort of. Actually, we've skipped over the hard part, which isn't about running scripts at all: you need to find a broker which is willing to accept and redistribute the events you send them to the network. In general, broker owners (or, at least, the responsible ones) will be conservative about who they give access to. There are good reasons for this: nobody wants to be responsible for acting as a gateway for useless events which will congest the network.

Once you have found a broker which is willing to accept your events, you can simply tell `comet-sendvo` about it with the `--host` option (you can also specify `--port` if necessary, but generally it isn't):

```
$ <voevent.xml comet-sendvo -h my.friendly.broker.org
```

For demo purposes, rather than spewing useless events onto the network, let's run our own broker. Comet makes this pretty trivial. Open a new terminal and run:

```
$ twistd -n comet -r --local-ivo=ivo://hotwired.org/test
```

The `-r` instructs Comet to listen for submissions from authors. We've not told it to either act on events it receives or to distribute them to subscribers, so it won't actually do anything with the events it receives, but we can, at least, demonstrate that the transmission works.

In your original terminal, go ahead and send the event to the broker running on `localhost`:

```
$ <voevent.xml comet-sendvo --host=localhost
2015-04-17 19:26:44-0400 [-] Log opened.
2015-04-17 19:26:44-0400 [-] Starting factory <__main__.OneShotSender instance at 0x1025743b0>
```

```
2015-04-17 19:26:44-0400 [INFO VOEventSender,client] Acknowledgement received from IPv4Address(TCP,
2015-04-17 19:26:44-0400 [VOEventSender,client] Stopping factory <__main__.OneShotSender instance at
2015-04-17 19:26:44-0400 [INFO VOEventSender,client] Event was sent successfully
2015-04-17 19:26:44-0400 [-] Main loop terminated.
```

Meanwhile, at the broker end, you should see something like:

```
2015-04-17 19:26:44-0400 [INFO comet.utility.whitelist.WhitelistingFactory] New connection from IPv4A
2015-04-17 19:26:44-0400 [INFO VOEventReceiver (ProtocolWrapper),0,127.0.0.1] VOEvent ivo://hotwired
```

Watch out, though: if you try to send the same event again, you'll get a failure:

```
$ <voevent.xml comet-sendvo --host=localhost
2015-04-17 19:28:07-0400 [WARNING VOEventSender,client] Nak received: IPv4Address(TCP, '127.0.0.1', 8
2015-04-17 19:28:07-0400 [VOEventSender,client] Stopping factory <__main__.OneShotSender instance at
2015-04-17 19:28:07-0400 [WARNING VOEventSender,client] Event was NOT sent successfully
```

What's going on? Well, recall that brokers can subscriber to each other's outputs. What we don't want is a single event to circulate on the network indefinitely, being passed back and forth between brokers. For that reason, a broker will refuse to process an event that it has seen previously.

CHAPTER 7

Future challenges

As described, VOEvent and its surrounding infrastructure meets the twin goals of describing transient events in a machine readable format and of providing a relatively low-latency system for distributing them to end users. However, the system is not without its drawbacks, and there remain substantial challenges to be solved both in delivering results that are of the greatest scientific relevance and in scaling to address next-generation surveys. Here, we highlight just a few areas where we suggest more work is needed.

It's worth emphasizing that the aim here is not to dismiss the achievements of the VOEvent community to date, or question its viability in the future. Rather, see this as a call-to-arms, and please jump in when you have something to contribute.

7.1 Authentication and verifiability

We made a VOEvent. Further, we made VOEvent based on information we read on the web somewhere; it's not something we observed, or that we analysed ourselves. We were careful to indicate in that VOEvent that we had no authority to speak on behalf of Gaia, but there was no check, no requirement to prove our identity. If we'd written that we were the Gaia Photometric Science Alerts Team, nobody would have denied it. Of course, actually *persuading a broker to accept* our event might be harder, but that's entirely dependent on the owner of the broker.

So: are you going to trigger your billion Euro telescope based on something that *might* be a genuine alert from Gaia? Well, it's your call...

Broadly, there are two ways you might try to tackle this problem. One is to authenticate the transport mechanism: you know that this is a genuine event because you got it directly from Gaia. That's possible using VTP, since Gaia could run a broker and you could subscribe to it directly. It nullifies the advantages of the distributed VOEvent network, though, and it means that effectively every project distributing events will have to run their own broker, or find some way of delegating permissions.

An alternative is to authenticate the events themselves. Potentially, this could be done by applying a cryptographic signature to the events when they are generated. A couple of proposals have been made along these lines: one involving OpenPGP signatures (*Denny, 2008*) and one based on XML Digital Signature with X.509 certificates (*Allen, 2008*).

Unfortunately, neither of these systems have seen significant adoption by the community. Partly, that's because on the small scales we have today, the risk is minimal. That happy situation will not last. Further, both of the potential solutions are complex. Applying a signature to an XML document is a fragile process: XML Digital Signature attempts to make it robust by formalizing a complex "canonicalization" procedure which must be carried out prior to signing, while the OpenPGP approach adopts an ad-hoc series of contortions which is simpler but error prone. Neither approach is ideal. And that's even before we've started discussing how to set up an appropriate public key infrastructure...

7.2 Discoverability

VTP makes it *easy* to subscribe to a broker and get streams of VOEvents. But... which broker (or brokers) are you going to subscribe to? What events are they going to send you? How can you learn about new brokers, or about changes to old ones, or about other capabilities that brokers offer?

The brief answer is that, at the moment, there's no reliable way to do this. They best you can hope is that you might find something on the list of brokers on the TDIG website, but even that is rather hit-or-miss.

A potential solution to this is to invoke the IVOA Registry (*Demleitner, 2014*). Work on an extension to the existing registry standards to describe VOEvent and its supporting infrastructure is underway. But even from there, it's a long way to an approachable, user-friendly way of figuring out which resources are available to the end user.

7.3 Scalability of VOEvents and VTP

We *started* by discussing the upcoming "deluge" of tens of millions of events per day. This is far more data intensive regime than the one we see today, and it's not one that the existing infrastructure has particularly been designed to address. XML is a flexible but verbose: that means it's slower than strictly necessary both to transmit and to parse. VTP is a chatty protocol, which requires multiple network round trips to transmit a single VOEvent. Expand this to a distributed network where transmitting an event requires multiple hops, each with multiple round trips, and the latency will rapidly start climbing.

In fact, *Swinbank (2014)* shows that it's possible to sustain transmission of reasonably high event volumes using VTP and Comet under ideal conditions. Ultimately, though, a root-and-branch rethinking of the problem may be necessary.

At the same time, other, completely unrelated, projects have addressed technical problems that are, in some ways, similar to that faced by VOEvent. For example, consider the Twitter Streaming API or Pusher. They combine an approach which is demonstrated to scale to high volume with a more modern approach to protocol and API design. But... they look plausible in 2015. What about when all these surveys are going live in 2025? Do we need to anticipate what direction technology will have taken? Is the event format sufficiently decoupled from the transport layer that it doesn't matter?

7.4 XML as an event format

VOEvent is designed around representing transient events as XML documents. But is XML really the most appropriate format? It does provide structure, and a wide variety of publicly available tools. But it's an old technology, apparently well past its peak. It's *difficult to sign*, *verbose* and—crucially!—unpopular with developers. Should we consider adopting another format? JSON is one possibility, for example. But there's already infrastructure and tooling built around XML-based VOEvents, and we can more easily integrate with the wide range of existing IVOA services. Is the pain of switching worth it?

7.5 High level infrastructure and scientific understanding

It's obvious that the tools shown here are building-blocks for a higher level infrastructure: end users should not have to worry about the details of network protocols or XML parsing in order to carry out their science. How can we build upon VOEvent to increase our scientific productivity?

The challenges are various. How do you select the few events which might be relevant to your science case from the multi-million event deluge? How do you prevent all the follow-up telescopes from chasing the same target? How do you cross-reference detections from multiple telescopes at different wavelengths (or, indeed, using entirely different types of messengers). How do you present all this with a digestible user interface which is accessible to the astronomer who just wants to get their science done?

Bibliography

Allan, A. & Denny, R.B., *VOEvent Transport Protocol, Version 1.1*. International Virtual Observatory Alliance Note, August 2009. Allen, S.L., *VOEvent authentication via XML digital signature*. Astron. Nachr., 329, 298–300, March 2008. Demleitner, M. et al, *The Virtual Observatory Registry*. Astronomy & Computing, 7, 101–107, November 2014. Denny, R.B., *A Proposal for Digital Signatures in VOEvent Messages, Version 1.1*. International Virtual Observatory Alliance Note, August 2008. Derriere, S. et al, *An IVOA Standard for Unified Content Descriptors Version 1.1*. International Virtual Observatory Alliance Recommendation, August 2005. Poci, A. et al, *DESAlert: Enabling Real-Time Transient Follow-Up with Dark Energy Survey Data*, submitted to PASA, April 2015. Seaman, R. et al, *Sky Event Reporting Metadata, Version 2.0*. International Virtual Observatory Alliance Recommendation, July 2011. Swinbank, J., *Comet: A VOEvent Broker*. Astronomy and Computing, 7, 12–28, November 2014. Williams, R.D. et al, *Skyalert: Real-time Astronomy for You and Your Robots*, Proceedings of ADASS XVIII, ASP Conf. Ser 411, 115, 2009.