"WATERSLIDE Programming Zen"
How to develop processing elements in waterslide

Updated May 2015

1 Introduction

WATERSLIDE is an event-at-a-time processing architecture for processing metadata. It is designed to take in a set of streaming events from multiple sources and pass them through a set of processing functions (called kids) that return meaningful outputs. WATERSLIDE can process grouped events at a time by using state tracking data structures to turn events into event summaries.

So, somehow you acquired WATERSLIDE and want to know how to program for it.. Well this document is intended to give you some examples of how to develop processing elements and some guidance into capturing, specifying, and manipulating dynamic data within this framework.

2 Design Philosophy

Here are some design guidelines to follow:

The executable for WATERSLIDE (or WS, for short) is entitled 'waterslide'. The WATERSLIDE core architecture is coded in C for maximum portability across a wide variety of architectures and compilers. The code is written in an object-oriented style, but does not use C++ but instead relies on structs and function pointers. Processing elements and datatypes are dynamically loaded at initialization runtime to limit the need to compile. In general, the folks who are coding new processors are C programmers. You can also program processing functions/kids in C++ and templates are included in the source codes for coding in C++.

In general, all memory in WATERSLIDE is allocated, reused and never freed. Therefore, much of the data allocated for WATERSLIDE assumes fixed-length data structures that can easily be reused. The only exception is in tightly controlled, infrequent cases such as removing a running processor. The reason for this is that in a continuously running process, regular memory allocation and deallocation tends to fragment memory. Because WATERSLIDE is intended to run for long periods of time without resetting, WATERSLIDE development takes effort to make sure that memory is properly reused.

Data structures that store and maintain state are typically operating at continuous saturation - so expiration of old data is one of the design elements for each of the stateful data structures found in the libraries of waterslide.

WATERSLIDE allows for more generic processing functions/modules to be created that can do stateful processing of hashed, keyed data. This configuration allows data functions to be data-type independent. This also allows for data and functions to be specified and resolved at runtime.

3 The Data Model

There are many different critical types of data in WATERSLIDE. We will first cover the three types of data that have a relatively narrow scope. The rest of the data types are used to globally pass data between the kids.

3.1 Data local to each instantiation of a kid

Each kid is implemented as a function in C. Because metadata is passed from one kid to the next in a processing graph, a new instance of the kid function is called each time. Therefore, any data that is allocated inside of the kid function is local to each kid as it processes the metadata passed to it. This is useful for local calculations and manipulations of the metadata.

3.2 Data local to each kid but persistent between instantiations of the kid

Clearly there needs to be a technique for data to persist between calls of each kid. This is accomplished using a struct with each kid that is maintained by the core WS processing engine and this struct is passed back to each kid as it is called. This struct is typed as `proc_instance_t` and is defined in each kid. The standard variable name for this struct is 'proc' and often holds global data such as counters, state tables, flags, etc.

3.3 Data persistently shared between kids

There is also a technique to share persistent data between kids. An example of where this would be required is in shared state tables (e.g. lastn kid) and shared work queues (e.g. workbalance kid). These shared-data objects are particularly useful in threaded WS graphs, and techniques for using these objects will be described in a later section.

3.4 Data that passes between kids in a processing graph

In WATERSLIDE, the basic data element that is passed between the processing functions (kids) is a typed and labeled generic buffer of fixed length. This allows the architecture to map in any fixed data structure for processing. This core datatype is name `wsdata_t` and will be described in greater detail later. This fixed structure also has a reference to predefined dynamic datatypes. These datatypes are defined and allocated at initialization runtime using shared objects located in the src/datatypes directory. The fixed and dynamic datatypes are coded as individual objects, but are stored in C data structures instead of a C++ class.

3.4.1 Free queue associated with each datatype

Rather than rely on expensive system `malloc` and `free` calls to manage data, WS uses an internal memory manager to more efficiently manage data. Each datatype maintains a separate freelist of available elements. At runtime initialization, some number of each of the datatypes are preallocated. As kids request allocations for a datatype, WS provides the data from the datatype's freelist. WS will `malloc` more of any particular datatype, as needed. When kids release the datatype, WS puts the freed memory back on the free list for future use.

3.4.2 Using reference counters to keep track of data

As individual data may be needed by more than one processor (and to avoid having to copy the data for use by multiple processors), WS uses the concept of a reference counter to determine if an individual piece of data is still in use. Each piece of metadata (eg. `wsdata_t`) has a reference counter reflecting the number of dependent processors on that data. If the reference counter goes to zero, that data can be freed. When a piece of metadata is no longer needed, it is reclaimed and put back on the freelist of its datatype. This data freeing occurs when data is finished processing from a processing kid. The `wsdata_add_reference` function increments the reference counter and the `wsdata_delete` decrements the counter.

3.4.3 Explicit dependencies between data

WS uses explicit dependency tracking between data to determine when data can be safely returned to the freelist. The WS helper function `wsdata_assign_dependency(parent, child)` is used to tell the child element that it has an association with the parent element. It also increments the parents reference counter to indicate that another data element is using the parents data. When the reference count of the child is decremented to zero, and it is returned to the free list, the reference count(s) of the parent(s) will also be decremented.

3.4.4 Structures and relationships between WS data types

The main, fixed-sized data element processed by WS is `wsdata_t`, and consists of:

- a pointer to the actual data
- a pointer to a fixed-sized datatype structure that defines the type of data (see below)
- a reference counter (the number of copies/references to this data)
- a reference defining the associated datatype and functions for that datatype - printing, reallocation, etc.
- an optional set of labels to describe the data. Multiple labels can be added to a given datatype-following the "Everything is Miscellaneous" philosophy.
- a optional set of references to dependent data relationships
- references to the free list for the `wsdata_t` structures
- Miscellaneous mutex, pointer, and hash information

Each of the WS dynamic datatypes is defined in the src/datatypes/ directory. The naming convention for each type follows the notation, `wsdt_{typedef}_t`, where `{typedef}` is a name that represents the type of data (e.g. `wsdt_string_t`, `wsdt_uint64_t`, etc). For each of the defined datatypes there is also another fixed-sized structure that provides a characterization of each datatype and also maintains helper information for the datatype. The structure is called `wsdatatypes_t` and includes the name of the type (in string form), a hash of the name, the length of the data, pointers to the free list for this datatype, mutex lock control, an optional list of subelements for this datatype, and a set of helper function pointers for

- initializing new data
- printing the data in ASCII, binary, and other formats
- reading in data from a string and
- converting to other formats or
- serializing data for network messaging transport

The `wsdatatypes_t` structure is initialized at the beginning of the runtime for each defined datatype. A variable name in the form of `dtype_{typedef}` (where `{typedef}` is the name of the type, e.g. `dtype_string`) is also associated with each datatype, and this dtype variable is often used in many of the helper functions.

Every data element defined in WS consists of these three components: 1) the main `wsdata_t` structure that points to both the actual data structure, 2) an `wsdatatype_t` structure that provides information, and 3) helper functions for each type.

Another important concept with WS datatypes is that some types are simple enough that the `wsdt_{typedef}_t` structure refers directly to the data, while more complex datatypes will have pointers to data that must be allocated separately. For example, the `wsdt_uint64_t` is simply a `uint64_t` while `wsdt_string_t` is a structure that has both a char pointer to a buffer and an integer describing its length. Details for allocating data will be presented later.

3.4.5 Useful operations for datatypes

Rather than dealing individually with each of these components (e.g., memory allocation, management, reuse, etc.), a WS programmer should use the helper functions provided in the WS tools to manipulate data. Not only does this make working with data much easier, it also ensures that data is created, used, reused, and destroyed in a consistent and efficient manner. This next section will describe many of the functions for manipulating data.

3.4.5.1 Data allocation

The basic data allocation function is:

```
wsdata_t * mpstr = wsdata_alloc(dtype_string);
```

Notice that this function will return the wsdata_t structure pre-filled with the information associated with the string type. In this case, the `wsdata_t`, `wsdatatype_t`, and the `wsdt_string_t` structures are all allocated and returned. Also, take special note that the actual string buffer and its length are initialized with NULL and 0 and must be filled in later. A `dtype_uint64` would actually have a `uint64_t` element available. In order to access the data pointed to by the `wsdata_t` variable, a common method is to define a variable with the structure associated with the type of data in the `wsdata_t`. For example, to reference the string created above:

```
wsdt_string_t * str = (wsdt_string_t *) mpstr->data;
```

You can now reference the data in the string using `str.buf` and `str.len`.

One particular, and very useful, allocation function creates a buffer of size MAXLEN and properly points the `wsdt_string_t` structure to a `buf` and `blen` you provide:

```
wsdata_t * mpbin = wsdata_create_buffer(MAXLEN, &buf, &blen);
```

Although these types of allocations are useful, the much more common method of allocating data is as a member of a tuple datatype (`wsdt_tuple_t`). This will be explained in much more detail after we have introduced the tuple datatype in a later section.

Remember that data is normally allocated from a free list maintained for each datatype.

3.4.5.2 Freeing data and the data reuse model

When a piece of metadata is no longer needed, it is reclaimed back into the free list of its datatype. Each piece of metadata has a reference counter assigned to the number of dependencies to that data. If the reference counter goes to zero, that data will be freed. This data freeing occurs when data is finished processing from a processing kid or when manually deleted. If that data has a forward dependence on another piece of data that forward dependency reference will also decrement upon data deletion.

3.4.5.3 Creating a new datatype

You can easily define your own datatype for processing by adding your own wsdt header and c files in the src/datatypes directory. There is also a template available for generating a new data type (e.g., wsdt_template.h and wsdt_template.c). A datatype can be created with any static structure, a set of printing functions and an optional set of functions for initializing (or reinitializing) and hashing data.

3.4.5.3.1 Default functions and user-defined functions

There is a default set of functions that apply to a user-specified data structure. In most cases, the defaults for initializing and deleting a data structure are likely to be sufficient. Additionally, the print function is another function that will most likely be needed.

3.4.5.4 Important files

Some important source files are discussed below, along with their use:

src/lib/wstypes.c initializes each of the data types

src/lib/waterslidedata.c `wsregister_label_internal`, label registration and manipulation, `wsdatatype_register`

src/include/wstypes.h provides types of all datatypes, sets up conversion functions (`dtype_get_uint32`, strings, etc.) and provides `wsdatatype_t * dtype_{all data types}`, and allocates buffer and string types

src/include/waterslide.h defines the `wsdata_t` structure, `wsdata_add_label`, `wsdata_check_label`, `wsregister_label`, `wsregister_label_wprefix`, threadsafe reference checking and manipulation, label set definition, etc.

src/include/waterslidedata.h defines the `wsdatatype_t` structure, subelement functions, `wsdata_core_init`, and `wsdata_alloc`

src/datatypes/wsdt_[type].[c,h] defines the actual datatypes and their helper functions

3.5 Tuples

Every stream processing framework has a concept of a 'collection' of data. In WATERSLIDE a dynamic collection of data is called a tuple. A tuple is a searchable container/array of multi-typed data. One of the big debates in developing WATERSLIDE was the issue of dynamic vs static tuples. A static tuple is one where the data exists in a specific location and is strongly typed. Static tuples have a nice advantage of being easy and efficient for locating subitems or members since these members are likely to be just offsets at a fixed index. In a dynamic tuple, members could be in arbitrary locations and searching is required to locate items. Dynamic tuples can be advantageous because they allow data to be abstracted, grouped, ordered, and labeled as needed for processing without changing the data structure. In addition, dynamic tuples allow programs to easily append arbitrary data. Since WATERSLIDE is a research framework for exploring stream data processing, we have chosen to standardize on dynamic tuples. This does not mean that WATERSLIDE does not have support for static tuples - in fact there are a number of static data structures within WATERSLIDE that are used to process standard fixed formats. The dynamic tuples in WATERSLIDE provide a nice data abstraction that allows generic operations for processing a wide-array of growing datatypes.

'Tuple' is the name given to the datatype used in WATERSLIDE that provides a dynamic list of typed, labeled data to be stored in a single container. A processing kid can easily and conditionally append data to a tuple. This allows a processing kid to add annotations, translations and summarizations of processed and extracted data. There are several processing kids that take in raw data, extract out interesting metadata, and output a set of tuples that can be processing by other tuple-friendly processing kids.

Tuples provide flexibility in the design of processing kids such that kids work with various datatypes of data transparently and independent of the position or location of the member element in the tuple. It is common for a user of a processing kid to specify the members of the tuple to process by specifying the label of a tuple member. That way the same processing kid can process buffers from many different data sources without having to know the overall data structure or other domain specific data structures.

We have many modules that can process on specific labels or datatypes present in a tuple.

Tuple data can be sub-selected, replicated easily through a set of tuple API's and existing processing modules.

3.5.1 Limitations

Data within a tuple must be searched to find member-data of the tuple. Searching is done by matching labels or datatypes. There are no fixed locations or predefined spaces allocated for a given datatype or label within a tuple. So, with the advantage of creating a flexible data object, we also have the penalty of searching to find data within a tuple. There is no limit on the number of additional data objects that can be added to the tuple.

3.5.2 Adding data in a tuple

There are several ways to add data to a tuple.

One way to allocate members of a tuple is, within the processor source code, to include wstypes.h, allocate a set of labels in the `proc_init` function using the `wsregister_label` function and then use `tuple_member_create` to create a new datatype within a tuple. For instance, to add two new labels to a tuple, MYLABEL1 and MYLABEL2, the following lines would be added to the `proc_init` function:

```
//Note: the proc->label_mylabel<1,2> variables should be declared in the
// proc_instance_t struct and  must be of type, "wslabel_t*"
proc->label_mylabel1 = wsregister_label(type_table, "MYLABEL1");
proc->label_mylabel2 = wsregister_label(type_table, "MYLABEL2");
```

Then, in your processing function, the following lines would add the actual data values to the tuple with the previously registered label.

```
uint32_t myuint = 32;
wsdata_t * wsd_myuint =
        tuple_member_create_uint(tuple, myuint, proc->label_mylabel1);

wsdt_fixedstring_t * mystr = tuple_member_create(tuple,
                                                 dtype_fixedstring,
                                                 proc->label_mylabel2);
if (mystr) {
        fixedstring_copy(mystr, "FOO", 3);
}
```

Another way to add data to the tuple is to add already allocated data to the tuple. To do this, use the `add_tuple_member` function to assign data to the tuple. With these tuple addition methods, the tuple helper functions make sure that the proper data dependencies are set so that data is not freed or reused improperly.

If you want a new string or binary buffer, there are helper functions to allocate new buffers and assign all the correct dependencies. These are `tuple_create_string` and `tuple_create_binary`. These functions take in a tuple, an optional label, and a desired length. If a buffer can be created to hold that length, then that buffer is added to the tuple and a pointer to a string or binary type is returned. This is a handy handy way of allocating buffers of arbitrary sizes.

If you are adding an integer or double member to a tuple, you can use the `tuple_member_create_uint`, `tuple_member_create_uint64`, or `tuple_member_create_double` functions to quickly add a numeric member to a tuple.

It is always good to make sure a `tuple_member_create` function successfully created a member. If the tuple is full or if the system is out of memory, the `tuple_member_create` function will return NULL. For reference, there is an extensive library of `tuple_member_create` functions in the wsdt_tuple.h header file located in the datatypes directory.

3.5.3 Searching within a tuple

In the wsdt_tuple.h header there are functions for initializing and performing label searches on a tuple. Each tuple keeps a hash table containing each label to enable fast searching, and to search for a member in a tuple, you need its label. When searching for a label in a tuple, the search returns

a list of members in the tuple that match that label (i.e., each tuple can have multiple members sharing a label).

For example, if a processor was being developed that used values in the DATETIME and NAME labels of a tuple, the DATETIME and NAME labels are first initialized in the `proc_init` function, as shown here:

```
proc->label_date = mpsearch_label(type_table, "DATETIME");
proc->label_name = mpsearch_label(type_table, "NAME");
```

Then, within the processing function, the tuple is searched for the data contained by these labels. Below is an example of performing this search:

```
    //datatypes are declared for holding pointers to the data.
    wsdt_ts_t * ts = NULL;
    wsdt_string_t * name = NULL;

    //now, declare datatype pointers that will point to the actual data
    wsdata_t ** mset;
    int mset_len;

    //find the specified label in the tuple (input_data), along with
    // size of the wsdata_t
    // if successful, this will point to an array of wsdata_t that match
    // the specified label.
    if (tuple_find_label(input_data, proc->label_date,
                              &mset_len, &mset)) {
            if (mset_len) {
                    //choose the first ts match
                    ts = (wsdt_ts_t*)mset[0]->data;
        }
    }

    //Now, do the same thing for the NAME label.
    if (tuple_find_label(input_data, proc->label_name,
                              &mset_len, &mset)) {
            if (mset_len) {
                    //choose the first match
                    name = (wsdt_string_t*)mset[0]->data;
        }
    }
```

Alternatively, the `tuple_find_single_label` function will search for a label in a tuple, but simply returns the first `wsdata_t *` that matches.

3.5.4 Tuple member datatypes - strings and numbers

Since WATERSLIDE has several datatypes that can contain content buffers, it is necessary to write routines that can verify and extract out the correct buffer pointer and length from these various structures. The helper function `dtype_string_buffer` in src/include/wstypes.h helps in the datatype validation and extraction of content buffers. Given a `wsdata_t` object, it checks known types and extracts out content buffers. The `dtype_string_buffer` function returns true if a datatype was found and pointers are returned that point to the buffer and length of the buffer.

Similar functions exist to extract out data from numeric types. These are `dtype_get_uint` and `dtype_get_double`. These functions return true if the datatype can be parsed and data at a pointer to the integer or double is filled in with the decoded value.

3.6 Flush events

In addition to tuples, WATERSLIDE also has a *flush* datatype. The two types of flush events, exit flush and inline flush, are used to flush state tracking structures internal to kids in the processing graph. When a processing graph completes, or is interrupted (e.g., by pressing `Ctrl+c`), an exit flush is issued to each of the kids (that have a registered flusher) in the processing graph, which then process the flush event based on their internal functions. Some issue processing statistics, some flush internal tables, and others exit silently. In contrast, certain kids (e.g., `flush`) issue inline flushes to the processor immediately downstream, causing that kid to flush internal state tracking structures. In most cases, inline flushes are not forwarded beyond the kid immediately downstream from the flush generator.

When developing a processing module, you are responsible for building functions that process and, optionally, issue or forward inline flush events. Functions are included for recognizing flush events, handling exit flush events, and forwarding inline flush events:

```
static int proc_flush(void * vinstance, wsdata_t* input_data,
                        ws_doutput_t * dout, int type_index) {

     proc_instance_t * proc = (proc_instance_t *)vinstance;
     proc->dout = dout;

     if (!proc->flush_walker) {
          proc->flush_walker =
               stringhash5_walker_init(proc->my_table,
                                       flush_my_destroy,
                                       proc);
          if (!proc->flush_walker) {
               return 1;
          }
     }

     int emit = 0;
     int row = proc->flush_walker->row;
     int cnt = proc->flow_table->all_index_size;
     do {
          emit = stringhash5_walker_next(proc->flush_walker);
          cnt--;
     } while (!emit && (row < proc->flush_walker->row) && (cnt > 0));

     if (proc->fwdflush) {
          // distinguish exit flushes from inline/intermediate flushes as we
          // only forward inline flushes
          if(!dtype_is_exit_flush(input_data)) {
               ws_set_outdata(input_data, proc->outtype_flush, dout);
          }
     }

     return 1;
```

```
}
```

4 Creating a Kid: Anatomy of a Processing Module

Each metadata processor processes a single event at a time. This differs from other complex event processing architectures which often deal with sets of data at a time. In WATERSLIDE, the only sets that are processed are sets that are part of a single tuple or inherent to a user-defined event data. When state is needed to be kept, it must be kept locally as part of a processing element. A processing element only defines acceptable inputs, expected outputs, initialization, and teardown functions.

4.1 Initialization

When an instance of a processing module is initialized, a new instance data struct is created and populated based on command line options and default settings. This initialization is called only once for each instance of a processor. This is done in the `proc_init` function. Included in the template for a module (proc_template.c) is a call to `proc_cmd_options` which allows the user of a given processor to specify instance-specific command line options. In this way each module can have its own way of specifying user defined options.

Here is an example of a processing module instance data structure:

```
typedef struct _proc_instance_t {
     uint64_t meta_process_cnt;
     ws_tuplesearch_t * tsearch;
     ws_outtype_t * outtype_tuple;
     wslabel_t * label_out;
} proc_instance_t;
```

These values are initialized in the `proc_init` function. Below is an example of a processing module initialization function that creates a new instance of a processing module and initializes the values defined in the `proc_instance_t` struct:

```
int proc_init(mpkid_t * kid, int argc, char ** argv,
              void ** vinstance, ws_sourcev_t * sv,
              void * type_table) {

     //allocate proc instance of this processor
     proc_instance_t * proc =
          (proc_instance_t*)calloc(1,sizeof(proc_instance_t));
     *vinstance = proc;
     proc->tsearch = tuplesearch_create();

     //read in command options
     if (!proc_cmd_options(argc, argv, proc, type_table)) {
          return 0;
     }

     return 1;
}
```

4.2 Input/output

Each time input data is sent to the processor, the `proc_input_set` function is called to determine which processing function and what output will result from that input. The output of this function is a function pointer (i.e., the name of the function that will be called to process this data). The `proc_input_set` function will be called for each input instance, that is, it is generally called for each input type, port and label. In the `proc_input_set` function, you return the processing function to use for each datatype and labeled input port. As a result, you can have a single processing module that will accept different input types and call different processing functions depending on those types.

Below is an example of an input set function. Note that this processor will only process "TUPLE_TYPE" data, and will return NULL for any other datatype. The function that will be called when tuple data is found is called `process_tuple`.

```
proc_process_t proc_input_set(void * vinstance, wsdatatype_t * meta_type,
                              wslabel_t * port,
                              ws_outlist_t* olist, int type_index,
                              void * type_table) {
     proc_instance_t * proc = (proc_instance_t *)vinstance;
     if (wsdatatype_match(type_table, meta_type, "TUPLE_TYPE")) {
          proc->outtype_tuple = ws_add_outtype(olist, meta_type, proc->label_out);
          return process_tuple;
     }
     return NULL;
}
```

4.2.1 Input ports

If you have a processor that needs to be able to take in different streams of data and handle those streams separately, you can define a set of labeled ports for input. This allows a processor to accept the same input type but have a different processing function be assigned to a different port. To use ports, match appropriate port labels in the input_set function to your desired processing function for that port.

Below is an example of a more complex `proc_input_set` function that processes both TUPLE_TYPE and FLUSH_TYPE data. In the processing of the tuple, it also handles a NOT port for this processor.

```
proc_process_t proc_input_set(void * vinstance, wsdatatype_t * meta_type,
                              wslabel_t * port,
                              ws_outlist_t* olist, int type_index,
                              void * type_table) {
     proc_instance_t * proc = (proc_instance_t *)vinstance;

     if (wsdatatype_match(type_table, meta_type, "TUPLE_TYPE")) {
          if (wslabel_match(type_table, port, "NOT")) {
               return proc_not_tuple;
          }
          return proc_tuple;
     }
     else if (wsdatatype_match(type_table, meta_type, "FLUSH_TYPE")) {
          return proc_flush;
     }
```

```
    return NULL;
}
```

4.3 The processing functions

The processing function is a user-defined callback function used to take metadata, do processing and set the resulting output data for downstream processors. For each input data and optional port passed in the `proc_input_set` function, a processing function may be given as a function pointer. That function will then be tied to that datatype and port. For each call of a processing function, an single event data element is passed in as input as well as data structures for use in setting output data. It is up to the processing function to decide what to do with the data passed in - to calculate a summary, add additional metadata, convert the data to a new datatype, blindly pass the data or simply do nothing with the data.

4.3.1 Optimizing processing function calls

To keep WATERSLIDE optimized for processing data, it is desirable that processing functions be optimized to minimize conditional processing and function calls. To make code readable it is recommended that `inline` function calls be used when calling out to other functions within a processing function. Additionally, to keep the shared object functions clean and easily exportable, any function that is not a `proc_init`, `proc_input_set`, or `proc_destroy` but is part of a processing module and lives within the processing code should declared `static`.

4.3.2 Allocating data

The function, `mp_get_outdata`, allocates memory for a new data element. This allocation, in turn, notifies downstream subscribers of this data source coming from this processor, so it is feasible that this function can return NULL if no subscribers are specified to a given data type. To allocate data as a member of a tuple, you can use the `tuple_member_alloc` functions to automatically create new data and attach it to the tuple. However, if allocated data is not sent to the output, it must be deleted using the `wsdata_delete` function, otherwise it must be sent to the output using the `mp_set_outdata` to avoid a memory leak.

4.3.3 Setting output data

Use the `mp_set_outdata` function to emit data out from a processing function. This attaches data to its associated type-dependent downstream processing functions and also adds the data to the job queue. Note that this mechanism allows makes it possible to have multiple data outputs sent as output with just a single input.

4.4 The instance destroy function

When WATERSLIDE is stopped, each processing module's `proc_destroy` function is called for each instance of the module. This is where all memory allocated to a particular instance is freed.

4.5 Special class: a source

A source is a special class of processing module (kid) that accepts no input types and initializes itself as a source in the `proc_init` function. For each source, a source processor function is specified by the user and is called every time the source is polled for more data. The source is supplied a preallocated data structure based on the datatype specified at initialization time. The source processor returns a 1 (one) if more data is available or a 0 (zero) if the source has no more data. If all sources return a zero, waterslide will exit.

Here is an example of setting the processor as a source kid. This is done in the `proc_init` function:

```
//Note:
// -- 'outtype_tuple' is declared in this kid's proc_instance_t
// declaration as type "ws_outtype_t*".
// -- 'tuple_source' is the function pointer to the kid's function
//   for generating the data
// -- The 'sv' variable is passed into the proc_init function
proc->outtype_tuple =
     ws_register_source_byname(type_table, "TUPLE_TYPE", tuple_source, sv);
```

5 Documenting a Kid

As in any software project, documentation is crucial to facilitating use of your new processing module. The WATERSLIDE tool suite uses the `wsman` tool to provide users with documentation for each kid. The `wsman` tool is similar to `man` pages for Unix commands. If you run `wsman` without any arguments, it prints out a listing of all processing modules available to the user. If the user supplies `wsman` with the name of a kid, it will print out the available documentation for that module.

As a developer, it is your responsibility to ensure that your new kid is fully and effectively documented. The WATERSLIDE project does not have a formal style guide for documentation, but the following sections will help you complete a kid's documentation in the common style.

The following fields are available to the `wsman` program:

```
char proc_name[]
char *proc_tags[]
char proc_purpose[]
char *proc_synopsis[]
char *proc_description
proc_example_t proc_examples[]
char *proc_alias[]
char proc_version[]
char proc_requires[]
char *proc_input_types[]
char *proc_output_types[]
proc_port_t proc_input_ports[]
char *proc_tuple_container_labels[]
char *proc_tuple_conditional_container_labels[]
char *proc_tuple_member_labels[]
proc_option_t proc_opts[]
char proc_nonswitch_opts[]
```

By convention, you should include all documentation fields, even if they are empty. You can use the following `wsman` function to check the completeness of your documentation:

```
# replace <kidname> with the name of your kid
wsman -c <kidname>
```

5.1 Name

The `proc_name` field stores the name of the kid:

```
#define PROC_NAME "addlabelmember"
char proc_name[] = PROC_NAME;
```

5.2 Tags

The `proc_tags` field stores the functional categories to which the kid belongs:

```
char *proc_tags[] = { "profiling", "statistics", NULL };
```

The developer can assign any tags that seem appropriate to describe the kid and aid a user in locating the kid. The following are some suggestions:

- input
- output
- parser
- encoder
- decoder
- stream manipulation
- filtering
- graphing
- detection
- matching
- profiling
- statistics
- annotation
- math
- state tracking

Each kid may perform more than one function.

5.3 Purpose

The `proc_purpose` field stores a one-line description of the kid's functionality:

```
char proc_purpose[] = "Appends a new member to the tuple with a user-specified "
     "label and value";
```

5.4 Synopsis

The `proc_synopsis` field stores a basic description of the kid's usage:

```
char *proc_synopsis[] = { "addlabelmember <LABEL> [-E <ENV> | -V <value>]",
                          NULL };
```

The following notation is typically used in the synopsis:

angle brackets, <>, indicate replaceable arguments

square brackets, [], indicate optional arguments

vertical bar, |, separates choices

ellipsis, ..., indicates that the argument can be repeated

Use multiple strings if there are sets of options that cannot be used together. The synopsis should match the kid's options (see Sections 5.16 and 5.17).

5.5 Description

The `proc_description` field stores a detailed explanation of the kid's functionality:

```
char proc_description[] = "Appends a new member to the tuple with a user-"
     "specified label and value. Only one label can be added at a time. The label
        "
     "is assigned by the user; if no label is provided, the default is an empty "
     "string. The value of the new member is either (1) set to 'UNKNOWN'; "
     "(2) assigned by the user; or (3) extracted from an environment variable. "
     "New tuple members are always added at the end of the tuple. "
```

The description field can include new lines (`\n`) and tabs (`\t`).

5.6 Examples

The `proc_examples` field stores a collection of one-line samples of the kid's usage with explanation:

```
proc_example_t proc_examples[] = {
     {"... | addlabelmember SENSOR -E NAME | ...",
      "append a tuple member with the label 'SENSOR' and the value of the 'NAME'
         environment variable"},
     {"... | addlabelmember SEEN -V 5 | ...",
      "append a tuple member with the label 'SEEN' and the value '5'"},
     {"... | addlabelmember 'LOOKS BAD?' -V 'if you squint' | ...",
      "append a tuple member with the label 'LOOKS BAD?' and the value 'if you
         squint'"},
     {NULL, NULL}
};
```

5.7 Alias

The proc_alias field stores names that can also be used to invoke this kid:

```
char *proc_alias[] = { "alm", "addlabelm", NULL};
```

5.8 Version

The proc_version field stores the version number of the kid:

```
char proc_version[] = "2.4";
```

A kid's version number is not necessarily the same as WATERSLIDE's version number. The version number of a kid should be incremented with substantial changes to the kid.

5.9 Requires

The proc_requires field stores descriptions of the kids or input formatting required before using this kid:

```
char proc_requires[] = "DATETIME to exist in tuple";

// or if there aren't any requirements
char proc_requires[] = "";
```

5.10 Input Types

The proc_input_types field stores the list of datatypes that are accepted as input to this kid:

```
// if the kid only accepts tuple datatypes
char *proc_input_types[] = { "tuple", NULL };

// if the kid has a meta processing function for any datatype
// in addition to processing functions for specific datatypes
char *proc_input_types[] = { "tuple", "monitor", "any", NULL }

// proc_input_types are automatically set for procbuffer kids
// do not include a definition
```

See Sections 4.2 and 4.2.1 for examples of setting input types.

5.11 Output Types

The proc_output_types field stores the list of datatypes that are used as output from this kid:

```
char *proc_output_types[] = { "tuple", NULL };

// proc_output_types are automatically set for procbuffer kids
// do not include a definition
```

See Section 4.2 for an example of declaring output types.

5.12 Input Ports

The proc_input_ports field stores the list of the ports available for input to this kid:

```
proc_port_t proc_input_ports[] = {
  {"none", "check if item is in filter, set & pass if NOT already in filter
      (unique)"},
  {"DUPES", "set item in filter, pass if already in filter (duplicate)"},
  {"QUERY", "check & pass if item is NOT already in filter"},
  {"INVQUERY", "check & pass if item is already in filter"},
  {"SET", "set item in filter, pass no output data"},
  {"REMOVE", "remove item from filter, pass no output data"},
  {NULL, NULL}
};

// of if there aren't any ports
proc_port_t proc_input_ports[] = { {NULL, NULL} };
```

A port is used to invoke a specific functionality of the kid (see Section 4.2.1). Ensure that the input ports described in the documentation match the functionality programmed into the kid.

5.13 Tuple Container Labels

The proc_tuple_container_labels[] field stores the list of labels that will be applied to each tuple container during processing:

```
char *proc_tuple_container_labels[] = { "TUPLE", NULL }

// or if there aren't any tuple labels
char *proc_tuple_container_labels[] = { NULL };
```

The tuple output from the kid will include all of these labels. Ensure that the tuple container labels described in the documentation match the functionality programmed into the kid. See Section 3.5 for a description of how to register and add labels to a tuple.

5.14 Tuple Conditional Container Labels

The proc_tuple_conditional_container_labels field stores the list of labels that may be applied to each tuple during processing if the data in that tuple meets certain requirements:

```
char *proc_tuple_conditional_container_labels[] = { "DECODED", NULL};

// or if there aren't any conditional tuple labels
char *proc_tuple_conditional_container_labels[] = { NULL };
```

The tuple output from the kid may include zero or more of these labels. Ensure that the tuple container labels described in the documentation match the functionality programmed into the kid. See Section 3.5 for a description of how to register and add labels to a tuple.

5.15 Tuple Member Labels

The proc_tuple_member_labels field stores the list of labels that may be appended to the tuple as new members or applied to existing tuple members during processing if the data in that tuple meets certain requirements:

```
char *proc_tuple_member_labels[] = {
     "NAME", "PATH", NULL};

// or if there aren't any added member labels
char *proc_tuple_member_labels[] = { NULL };
```

The tuple output from the kid may include zero or more members with these labels. Ensure that the tuple member labels described in the documentation match the functionality programmed into the kid. See Section 3.5.2 for a description of how to register and add labels to a tuple member.

5.16 Options

The `proc_opts` field stores the options available to change the functionality of the kid when it is invoked:

```
proc_option_t proc_opts[] = {
     /* 'option character', "long option string", "option argument",
        "option description", <allow multiple>, <required>*/
     {'V', "", "value",
      "sets value of new tuple member", 0, 0},
     {'E', "", "ENV",
      "sets value of environment variable as tuple member", 0, 0},
     // the following must be left as-is to signify the end of the array
     {' ', "", "",
      "", 0, 0}
};

// or use long option strings
proc_option_t proc_opts[] = {
     /* 'option character', "long option string", "option argument",
        "option description", <allow multiple>, <required>*/
     {'F', "", "file",
      "load items to search from file (required)", 0, 1},
     // the following must be left as-is to signify the end of the array
     {' ', "", "",
      "", 0, 0}

};
```

The option's description should begin with a verb, as it describes the action enabled by invoking this option. The option's argument should match the synopsis (see Section 5.4). To aid searching by the user, organize the options in alphabetic order [numbers < letters (lowercase < uppercase)]. Ensure the options described in the documentation match the functionality programmed into the kid.

NOTE: The `<allow multiple> and <required>` fields of an option's documentation do not actually affect processing when a kid is executed. Additionally, they are not used in the kid's `wsman` page.

5.17 Non-Switch Options

The `proc_nonswitch_opts` field stores the inputs required (without an assigned option flag) when invoking the kid:

```
char proc_nonswitch_opts[] = "LABEL of string to parse"

// or if there aren't any non-switch options
char proc_nonswitch_opts[] = "";
```

Non-switch options are typically the LABEL(s) in the input tuple that will be searched for processing in the kid. The non-switch options should match the synopsis (see Section 5.4).

6 A Simplified Decoder - the procbuffer kid

WATERSLIDE includes many helper functions to make it easier to write a kid to decode a buffer. If you use this approach, you can utilize an alternate source code template without a lot of the boilerplate overhead of full-blown kids. This kid-writing approach can be particularly useful when writing file or protocol decoders.

The basic assumption with the helper library is that a developer is interested in processing a specific user-specified buffer that exists as a member of a tuple. Any output that resulted from the buffer processing would be appended information to the input tuple or to possibly filter the tuple based on the results of processing the buffer. Instead of writing all of the code to select and find buffers in a tuple, using a proc_buffer template a developer can focus only on writing code to process an already selected buffer.

The key function that needs to be specified in a procbuffer kid is:

```
int procbuffer_decode(void * vproc,
                      wsdata_t * tdata,
                      wsdata_t * member,
                      uint8_t * buf, int buflen) {

        // where vproc is a user specified instance state,
        //       tdata is the tuple event,
        //       member contains the user-specified buffer,
        //       buf is the location of the buffer in the member,
        //       buflen is the length of the buffer in the member

        //do your decode of buffer here
        //append items to tuple

        return 1;  //return 1 if tuple should be pass onward, otherwise 0
}
```

In fact, for this simplified form of a processor kid, no other function needs to be specified to make the kid run with proper defaults. However, because this is a special kid, a global variable needs to be set in the kid to signal the WATERSLIDE shared object loader to treat this kid as a decoder. Therefore, include the following line near the top of the kid as a global variable:

```
int is_procbuffer = 1;
```

A procbuffer kid will accept both tuple and monitor datatypes, and it will output tuple datatypes.

By default, a procbuffer kid will drop any input that cannot be processed (e.g., it does not include the expected LABEL). To change this behavior to allow all input to pass through the procbuffer kid, the following line must be included near the top of the kid as a global varialbe:

```
int procbuffer_pass_not_found = 1;
```

If the kid developer wants to register new labels for items to append in the tuple or accept command line options, you can also specify an instance struct that is passed to all kid functions. First specify a struct with variables that are passed to a kid instance, then specify the size of this struct for use when initializing the kid.

Here is an example struct and sizing information for use with a special proc_buffer kid:

```
typedef struct _proc_instance_t {
     wslabel_t * label_decode;
     int pass_all;
     char * ignore;
     int ignore_len;
} proc_instance_t;

int procbuffer_instance_size = sizeof(proc_instance_t);
```

This data struct is allocated for each instance of a kid, zeroed out, and passed as the first argument (i.e., the vproc pointer in the example above) to functions in the kid.

In the proc_buffer decoder kid, command line options are specified in a slightly different way than regular kids. The developer must specify a string with options and then a function for parsing those options. When the kid is initialized, these command line options will be parsed and a call to the supplied `procbuffer_option` function will be called for each option. Here is an example:

```
char procbuffer_option_str[]    = "s:L:p";

int procbuffer_option(void * vproc, void * type_table, int c, const char * str) {
     proc_instance_t * proc = (proc_instance_t *) vproc;

     switch(c) {
         case 's':
              proc->ignore = strdup(str);
              proc->ignore_len = strlen(proc->ignore);
              break;
         case 'L':
              proc->label_decode = wsregister_label(type_table, str);
              break;
         case 'p':
              proc->pass_all = 1;
              break;
     }
     return 1;
}
```

An optional `procbuffer_init` function can also be specified to allow the developer to register labels that will be applied to tuples or tuple members.

```
int procbuffer_init(void * vproc, void * type_table) {

     //allocate proc instance of this processor
     proc_instance_t * proc = (proc_instance_t *)vproc;

     if (!proc->label_decode) {
          proc->label_decode = wsregister_label(type_table, "DECODE");
```

```
    }

    return 1;
}
```

However, instead of including a `procbuffer_init` function, there is another, simpler way to register labels in the developer supplied `proc_instance_t` structure. You can use the `proc_labeloffset_t` structure to define a set of label strings and offsets into the structure. Instead of using the above `procbuffer_init`, you can instead simply place the following structure just after the definition of `proc_instance_t` with a structure:

```
proc_labeloffset_t proc_labeloffset[] =
{
    {"DECODE",offsetof(proc_instance_t, label_decode)},
    {"",0}
};
```

When using this option, the stddefs.h header must be included at the top of the proc_buffer source code to have the offsetof macro defined for your architecture. Also, note that the empty label (i.e., {"", 0})is necessary to signal the end of the labeloffset array.

The proc_buffer decoder kid type offers a simpler, alternative way of specifying a buffer decoder type kid by abstracting boilerplate tuple processing code. If you are looking for a template for writing a decoder, the proc_asciihex.c kid is suggested as a starting point.

7 A Simplified State Tracking (Keystate) Kid

In addition to the proc_buffer template and helper functions, WATERSLIDE includes helper functions to make it easier to write a kid to keep track of the state of specific elements by treating a user-specified member of a tuples as the key for state tracking. The state that is being tracked can optionally be updated by having the user of the kid specify a separate value of which can be processed to update the state. The intention of this simplified kid is to provide primitives for the reduce phase of a streaming map-reduce function. If you use this approach when generating the source code for a state-tracking kid, you can again generate the code without dealing with much of the boilerplate overhead of full-blown kids.

The basic assumption with the keystate helper library is that a developer is interested in keeping track of the state of user-specified keys (and optionally-user specified values) for processing tuples. The resulting state tracking function can be used to filter the tuple based on state or even wait until state expires before generating output tuples.

Since a state tracking kid needs to keep track of state, the developer needs to specify a struct that will be used to track the desired state. This struct will be allocated and individual values zeroed out for each new key arrives before it arrives. This structure can be specified in a similar manner to the following example:

```
typedef struct _key_data_t {
    uint64_t cnt;
} key_data_t;

int prockeystate_state_size = sizeof(key_data_t);
```

There are two other possible functions that might need to be specified in a keystate kid. These are:

```
// called after key is found in a tuple
// note that the state is already looked up and returned
int prockeystate_update(void * vproc, void * vstate, wsdata_t * tuple,
                        wsdata_t *key) {

     key_data_t * kdata = (key_data_t *) vstate;

     kdata->cnt++;

     return 1;  //return 1 if tuple should be pass onward, otherwise 0
}

// called after key and value are found in a tuple
// note that the state is already looked up and returned
int prockeystate_update_value(void * vproc, void * vstate, wsdata_t * tuple,
                              wsdata_t *key, wsdata_t * value) {
     uint64_t v64 = 0;

     //check if value is expected type
     if (!dtype_get_uint(value, &v64)) {
          return 0;
     }

     key_data_t * kdata = (key_data_t *) vstate;

     proc->totalcnt += v64;
     kdata->cnt += v64;

     return 1;  //return 1 if tuple should be pass onward, otherwise 0
}
```

No additional functions are required for a state-tracking kid to run assuming that proper defaults are supplied. However, because this is a special kid, a global variable needs to be set in the source code to signal the WATERSLIDE shared object loader to treat this kid as a keystate processor, instead of as a fully-developed kid. Include the following line near the top of the kid as a global variable:

```
int is_prockeystate = 1;
```

If it is desirable to register new labels and append new items to the tuple or accept command line options, you can also specify an instance structure that will be passed to all of the state-tracking kid functions. First, specify a struct with variables that are passed to each kid instance, then specify the size of this struct for use when initializing the kid. Here is an example of declaring a struct for a state-tracking kid:

```
typedef struct _proc_instance_t {
     wslabel_t * label_cnt;
} proc_instance_t;

int prockeystate_instance_size = sizeof(proc_instance_t);
```

This struct will be allocated for each instance of the state-tracking kid, variables zeroed out, and the entire struct will be passed as the first argument to functions in the kid.

In a keystate kid, command line options are specified in a slightly different way than regular kids. The developer must specify a `getopt`-style string with options along with a function for parsing those options. When the kid is initialized, these command line options will be parsed and a call to the supplied `prockeystate_option` function will be utilized for each option. Here is an example of option parsing for a keystate tracking kid:

```
char prockeystate_option_str[]    = "L:";

int prockeystate_option(void * vproc, void * type_table, int c, const char * str)
   {
     proc_instance_t * proc = (proc_instance_t *)vproc;

     switch(c) {
     case 'L':
          proc->label_cnt = wsregister_label(type_table, str);
          break;
        }
     return 1;  //return 0 only if option failed
}
```

An optional `prockeystate_init` function can also be specified to allow the developer to register labels that will be applied as tuple container labels or to tuple members.

```
int prockeystate_init(void * vproc, void * type_table) {

     proc_instance_t * proc = (proc_instance_t *) vproc;

     if (!proc->label_cnt) {
          proc->label_cnt = wsregister_label(type_table, "COUNT");
     }

     return 1;  //return 0 only if init failed
}
```

However, instead of creating and registering labels in a `prockeystate_init` function, there is a simpler way to register labels within the `proc_instance_t` structure. You can use the `proc_labeloffset_t` structure to define a set of label strings and structure members. See the preceding `proc_buffer` kid section for a discussion and examples of initializing labels using the `proc_labeloffset_t` structure;

Finally, if upon expiration of state, the developer wants events to be generated or if special cleanup is needed, you can specify an expire function to be called for each state that is about to be expired. This expiration occurs on a flush, when the program exits or when the state table is full. Here is an example expiration function:

```
void prockeystate_expire(void * vproc, void * vdata,
                         ws_doutput_t * dout,
                         ws_outtype_t * outtype_tuple) {

        //state about to be expired
     key_data_t * kdata = (key_data_t*)vdata;

        //kid instance that contained registered labels
        proc_instance_t * proc = (proc_instance_t *)vproc;
```

```
        //checking conditions for sending output on expire
    if (kdata->cnt) {
            wsdata_t * tup = wsdata_alloc(dtype_tuple);
            if (tup) {
                    tuple_member_create_uint(tup, kdata->cnt,
                       proc->label_cnt);
                    ws_set_outdata(tup, outtype_tuple, dout);
        }
    }
}
```

The `prockeystate_expire` function is optional and, thus, not required for a default keystate kid.

This simpler, alternative way of specifying a state-tracking kid should make it easier to add new code to WATERSLIDE. If you are looking for a template for writing a keystate kid, the proc_keyadd.c kid would be a suggested place to start.

8 The Processing Graph and Job Queue

On the command line or in a config file, the user may specify a processing graph that lays out a pipeline for processing metadata using several of the processing modules (kids) developed earlier. This pipeline is a true graph that starts with data generated from sources that propagates and is enriched along directed edges of the graph between other processing elements. In turn those processing elements take input data and generate or enrich data that propagate along subsequent edges. This is repeated until there is nothing left to process. To instantiate this graph, a `Job Queue` is used to enumerate the remaining data and processing stages left. Each processing element can add jobs (data + downstream processor functions) to the job queue. Additionally, the processing graph can contain multiple sources that are periodically polled for more data. The polled data is then added to the job queue and executed through the affected processing elements in the graph until completion. While it is possible to poll sources before running prior data to completion, the current WATERSLIDE implementation implements processing as an event at a time, and each event is processed through the entire graph before new data is polled and added to the job queue.

8.1 Building a processing Graph

At initialization of WATERSLIDE, the user-specified processing graph is compiled and then each processor in the graph is instantiated and initialized. This graph is built by specifying sources, processing elements and stream-variables connecting processing elements.

Here is an example processing graph that takes in data from a comma separate input file and finds repeated pairs:

```
#reads in csv files
csv_in FIELD1 FIELD2 -> $cin

$cin | tuplehash FIELD1 FIELD2 -L PAIR -> $pair

$pair | DUPES:uniq PAIR | removefromtuple PAIR -> $dupes

#print out metadata
```

```
$dupes | print -TV
```

The pipe, '|', command is used to specify a stream relationship where output data from one processing function is set as input to the subsequent processing function. This allows for easy specification of pipelined processing. In addition, it is possible to have an output sent to multiple processing functions through the use of stream variables (the elements in the graph that start with '$').

Stream variables are used to link streams of data from modules into one, or more, other modules.

A single processing function can have multiple inputs by specifying multiple stream variables on the initial left-hand side of a configuration line.

Each processing module can have its parameters specified at the command line. For instance, the tuplehash module accepts label names and other parameters as input to determine what to hash and how to label that hash function.

8.2 the Job Queue

Data needing processing is added to a global job queue. Each job in the job queue consists of metadata along with a set of subscriber processing functions for that metadata. Currently, when a graph is being processed, every source is polled once. The sources add jobs to the job queue, and the subscribing processors are executed on the data adding their output to the job queue. Each job is a processor executing on its subscribed data and adding any output data to the job queue. This process is continued until no jobs are left on the job queue. Then, the process repeats by polling the sources again.

9 Core Libraries

9.1 Data Structures

WATERSLIDE has several interesting data structures and libraries to help in algorithm development including queues, hash tables, expiring hash tables, bloom filters and variants, and search trees. Here is a brief description of several of these elements:

9.1.1 stringhash5

Stringhash5 is an expiring hash table for use in tracking information for as long as reasonably possible. Stringhash5 is found as one of the several header libraries in the src/include directory. It is kept as a header library so that critical sections of data can be put as inline functions into processing elements.

Stringhash5 is an expiring hash table with a localized-index and a least-recently accessed expiration mechanism. Upon initialization, the user specifies the table size (i.e., the number of records that can be included in the table) and the size of the datatype that will be placed at each key. The key can be of arbitrary length. Upon expiration, an optional callback function can be called to allow for user-specified graceful expiration. Data is reused at expiration rather than freed.

Stringhash5 has been designed to be implemented in shared memory where regions of the hash table can be locked without having to lock the entire table. Unlike a global least-recently-used mechanism (such as those found in stringhash3), stringhash5 uses localized sequencing information to decide which records to expire.

Some stringhash5 library functions are described below:

`stringhash5_create` create a new hashtable

`stringhash5_destroy` destroy and frees memory

`stringhash5_find` locate a record

`stringhash5_find_attach` locates and/or allocates record

`stringhash5_delete` remove a record

`stringhash5_scour` walk the hashtable, calling a callback function

`stringhash5_flush` zeroes the hashtable

`stringhash5_set_callback` set the function to be called when a record expires

9.1.2 stringhash9a

Stringhash9a is an expiring existence check table that operates much like an expiring bloom filter. Keys are looked up and the result is either a 1 for existence or a 0 for non-existence. There is also a check and set operation. It uses a multi-way localized least-recently accessed expiration mechanism. This library also has the ability to save state to a file and load state from a file.

Some stringhash9a library functions are described below:

`stringhash9a_create` create a new hashtable

`stringhash9a_destroy` destroy and frees memory

`stringhash9a_set` add a record to the table

`stringhash9a_check` check if a record exists in the table

`stringhash9a_delete` remove a record

`stringhash9a_flush` zeroes the hashtable

10 Debugging

WATERSLIDE has a number of useful ways to debug what is going on.

10.1 Debug printing

When debugging modules it may be useful to have very verbose output. One way to output information is to use `dprint` statements for debug-only output. To turn on debugging, one needs to specify:

```
#define DEBUG 1
//must be above
#include "waterslide.h"
```

This is typically placed at the top of a processing kid source code (above the waterslide.h include line). If this option is turned on, the `dprint` statements will print to standard error. If this option is not specified, the `dprint` statements will be completely ignored by the compiler. Using `dprint` to annotate code is useful in debugging, but can be easily turned off once the processing kid is ready for release.

10.2 Verbose output

By default WATERSLIDE reports only on processing kid status, but you can force WATERSLIDE to report more detailed status by specifying the verbose option (-V) on the main waterslide command line to force waterslide to report a more verbose output for both startup and shutdown data. For example:

```
#report a verbose output
cat listofcsv.txt | waterslide -V "csv_in | testtool"
```

Here is an example of a datatype-memory report that can be examined at the end of a run:

```
dtype UINT64_TYPE: allocd 2416 recovered 2416
dtype TUPLE_TYPE: allocd 2497 recovered 2497
dtype FIXEDSTRING_TYPE: allocd 11684 recovered 11684
dtype LABEL_TYPE: allocd 91 recovered 91
dtype UINT_TYPE: allocd 7278 recovered 7278
dtype FLUSH_TYPE: allocd 1 recovered 1
dtype STRING_TYPE: allocd 1517 recovered 1517
dtype MEDIUMSTRING_TYPE: allocd 3604 recovered 3604
dtype BINARY_TYPE: allocd 17700 recovered 17700
dtype TS_TYPE: allocd 2426 recovered 2426
```

Notice that this listing shows the number of allocated data structures and the number that have been recovered as a result of exiting. When there is a difference between the allocated and recovered memory, WATERSLIDE will report these results as an error.

10.3 Hashtable summary

When using the "stringhash table summary" option (-t), WATERSLIDE will report the memory space required for each hashtable employed in your processing graph. For example:

```
# report hashtable memory footprint
cat listofcsv.txt | waterslide -t "csv_in FOO | uniq FOO | print"
```

The following hashtable-memory report corresponds to the above pipeline:

```
WS Table Summary:

Local Tables:
rank:    0, type:          sh9a, kid:     uniq, size:    134217808 bytes

Local Tables, number: 1
                     global size:   134217808 bytes
```

Because this report includes only the memory necessary for each hashtable, you will still need to leave additional memory for the data being passed through the event queue. If your pipeline is

using too much memory for the execution environment, you can investigate re-sizing the offending hashtables.

A "WS Table Summary" report can be generated for any combination of shared and local memory hashtables appearing in a processing graph. For example, consider a multithreaded version of the previous graph, with a call to the `fixedmatch` kid added to invoke some local hashtables:

```
# report shared hashtable memory footprint
waterslide-parallel -t -F waterslide_input_p4.txt -F waterslide_test_shared_p4.txt
```

10.4 GDB

10.4.1 Basic Usage

In my Unix-like environments, the most commonly-used debugger is GDB. To load waterslide in GDB, simply run:

```
gdb waterslide
```

We can set breakpoints on different functions or lines with the "break" command:

```
// break at the start of the main function
(gdb) break main
Breakpoint 1 at 0x409c60: file waterslide.c, line 135.

// break when line 16 of waterslide.c, the print_cmd_help function, is
// encountered
(gdb) break waterslide.c:16
Breakpoint 2 at 0x409ae1: file waterslide.c, line 16.
```

Run the waterslide executable as normal with the "run" command. Any arguments following the run command will be interpreted command line arguments to the executable (for instance, 'run -F "myscript.ws"' would run the configuration graph coded in the myscript.ws file). For this example, we're simply going to display the waterslide usage information, so we use "run -h".

```
(gdb) run -h
Starting program: /home/wsuser/src/waterslide-core/waterslide -h

Breakpoint 1, main (argc=2, argv=0x7fffffffdb18) at waterslide.c:135
135     int main(int argc, char ** argv) {
```

We see that we hit the first breakpoint, which is the start of the main function.

A common debugging operation is to simply print the contents of a variable. GDB has the print command for this purpose. For example, we can easily print the command line arguments:

```
(gdb) print argv[0]
$1 = 0x7fffffffde4e "/home/wsuser/src/waterslide-core/waterslide"

(gdb) print argv[1]
$2 = 0x7fffffffde76 "-h"
```

To resume execution after hitting a breakpoint, use the continue command:

```
(gdb) continue
Continuing.
```

```
Breakpoint 2, print_cmd_help (fp=0x3632d8d860) at waterslide.c:16
16          status_print("waterslide");
```

After continuing, we hit the second breakpoint. We simply continue execution, waterslide prints the usage information and completes execution.

```
(gdb) continue
Continuing.
waterslide
 Version 1.0.0
  [-F <file>] load processing graph
  [-l <cnt>] loop graph n times
  [-L <file>] file capturing stderr output
  [-X] turn off flushing of kids
  [-s <seed>] set random seed
  [-G <file>] save graphviz graph
  [-C <path>] set config path
  [-D <path>] set datatype path
  [-P <path>] set procs path
  [-A <file>] set path + filename to alias (.mpalias)
  [-p <path>] set PID file path
  [-V] Verbose status printing
  [-v] run with Valgrind - keep shared objects at termination

Program exited with code 0377.
```

This is only a tiny subset of GDB's capabilities as a debugger. Refer to the official documentation to learn more about GDB.

10.4.2 Reading from standard input

It is common to read from standard input before beginning a WS pipeline. For example, the csv_in kid reads filenames from standard input. This presents a challenge when debugging WS in GDB because GDB is not a fully-featured shell.

A relatively simple workaround is to use file redirection. For example, suppose your pipeline begins with the csv_in kid. Instead of using the UNIX find or ls command to send filenames to waterslide, first generate the list of filenames, then redirect this list to waterslide within GDB. For example:

```
$ find /home/data/ -name '*.csv' > csv_files.txt
$ gdb waterslide
(gdb) run 'csv_in | print -V' < csv_files.txt
```

10.4.3 Debugging crashes with core files

Suppose you're writing a new processing kid called my_new_kid, and waterslide crashes for some unknown reason. Without special insight, it's difficult to diagnose the error. In Unix-like operating systems, a file containing the state of a program is created if it crashes. This is called a core file, and GDB can examine these files to gain insight into specifically why a program crashed.

The first step configure the maximum core dump size to ensure waterslide creates a core file when it crashes:

```
ulimit -c unlimited
```

Now we run the pipeline that caused waterslide to crash.

```
cat csv_files.txt | waterslide 'csv_in| my_new_kid | print -V'

initializing sources
[1]    16541 done                                cat csv_files.txt |
       16542 segmentation fault (core dumped)    waterslide 'csv_in| my_new_kid |
```

We see that a segmentation fault occurred and a core file was dumped. We need to find the name of the core file:

```
$ ls core*
core.16620
```

Now load GDB with the core file and our waterslide executable. GDB shows that the crash occurred on line 162 of the `print_process` function of proc_my_new_kid.c.

```
$ gdb -c core.16620 waterslide

Core was generated by `waterslide csv_in| my_new_kid | print -V'.
Program terminated with signal 11, Segmentation fault.
#0  proc_process (vinstance=0xdffaf0, input_data=0xe00c10, dout=0xdeca08,
type_index=0) at proc_my_new_kid.c:162
162          *ptr = 123;
```

We can use backtrace command to see the functions that executed prior to the crash:

```
(gdb) backtrace
#0  proc_process (vinstance=0x986af0, input_data=0x987c10, dout=0x973a08,
          type_index=0) at proc_my_new_kid.c:162
#1  0x0000000000042a635 in ws_do_local_jobs (mimo=<value optimized out>,
          jobq=0x8c6570, jobq_freeq=0x8c65a0) at wsprocess.c:90
#2  0x0000000000042aaa4 in ws_execute_graph (mimo=0x8c6010) at wsprocess.c:241
#3  0x000000000040a03f in main (argc=<value optimized out>, argv=<value
optimized out>) at waterslide.c:186
```

We can see the crash occurred on line 162 but it's not clear why this is an error without any context, so we can print the source code of the `print_process` function:

```
(gdb) list proc_my_new_kid.c:proc_process
157     static int proc_process(void * vinstance, wsdata_t* input_data,
158                             ws_doutput_t * dout, int type_index) {
159          proc_instance_t * proc = (proc_instance_t*)vinstance;
160
161          int * ptr = NULL;
162          *ptr = 123;
```

And we see the error occurred because we tried to modify the contents of a NULL pointer on lines 161 and 162.

10.5 Valgrind

In order to examine memory and look for invalid references to memory, the Valgrind tool is very handy. It can be applied to running code to check for memory leaks. Here is an example:

```
cat listofcsv.txt | valgrind --tool=memcheck waterslide "csv_in | testtool"
```

Valgrind will report memory error conditions and will report on the amount of memory allocated and recovered. Valgrind is slow, so it is suggested that only a small, representative subset of data is run through waterslide for testing.

11 Conclusion

WATERSLIDE is an event-at-a-time processing architecture for processing metadata. The executable for the WATERSLIDE framework is entitled 'waterslide'. This document has outlined the framework, structures, datatypes and processing elements and paradigms used within WATERSLIDE and is meant to serve as a basic guide to developing new processing functions, or kids, within this framework. Several simplified template approaches for new development of buffer-decoding and state-tracking were also outlined. Additional documentation for understanding the WATERSLIDE job queue, expiring hash tables, and debugging processing pipelines were also presented. Because the WATERSLIDE environment is rich, with over 80 processing kids, it is difficult to provide exhaustive documentation for methods and procedures for new development, the purpose of this guide is to provide a basic foundation and an entrance into this development environment.