CS 5412 - Cloud Computing

Recitation 02/21: Derecho

Sagar Jha

1 Derecho

We are first going to see how Derecho achieves SMR in terms of how the state is represented and how the commands are executed.

1.1 Application structure and State representation

Derecho manages application structure into subgroups. Think of subgroups as application components, for instance, a subgroup for Cache, Storage etc. Each one of the subgroups is further divided into shards (horizontal partitions). Every shard has a state, represented by a C++ class object. All members of a shard maintain the same copy of the object. Can you now guess how the commands are represented? The commands are member functions of the class. The user can call these functions to change the state for every replica of a given shard. All members in the system are part of a top-level group and node failures, joins and leaves of top-level group or subgroup shards are handled at the top-level.

More specifically, the user of Derecho writes C++ classes (called types here). Each type can have multiple subgroups associated with it. The user also writes a membership allocation function which maps every shard of every subgroup down to a vector of members. The membership allocation function takes the top-level group membership as its input. When should we run the allocation function? Recall that the top-level group membership is fixed in a view and changes between views. Thus, the allocation is done in a view change. This sets up the application structure before the next view starts. What does that mean? Every node in the top-level group gets replicated objects for the shards it's a member of. Nothing needs to be done if the node was already a member of the shard in the prior view because it will have a replicated object already. If the shard is newly created (it did not exist in the prior view), the object is freshly constructed by the class constructor. Otherwise, if the node joined the shard, it will get the state from an existing member via state transfer.

Below, we see some examples of how a type looks like:

```
/**
 * Example replicated object, containing some serializable state
    and providing
 * two RPC methods. In order to be serialized it must extend
    ByteRepresentable.
```

 */
```
class Foo : public mutils::ByteRepresentable {
    int state;

public:
    int read_state() {
        return state;
    }
    bool change_state(int new_state) {
        if(new_state == state) {
            return false;
        }
        state = new_state;
        return true;
    }

    REGISTER_RPC_FUNCTIONS(Foo, read_state, change_state);

    /**
     * Constructs a Foo with an initial value.
     * @param initial_state
     */
    Foo(int initial_state = 0) : state(initial_state) {}
    DEFAULT_SERIALIZATION_SUPPORT(Foo, state);
};
```

Foo is the type. The state of Foo is a single integer *int state*. The functions for changing the state is *bool change_state* (*int new_state*). Additionally, Foo has a read-only function *int read_state* (). How does Derecho know about these functions? The user, through a macro, *REGISTER_RPC_FUNCTIONS(Foo, read_state, change_state)* registers the class functions with Derecho. Derecho uses this information for serializing/deserializing function calls (more in detail, later).

Similarly, the code below shows the type Bar which has a state *int log* and functions *void append* (*const string& words*), *void clear* () and *string print* () for mutating or reading the state.

```
class Bar : public mutils::ByteRepresentable {
    std::string log;

public:
    void append(const std::string& words) {
        log += words;
    }
    void clear() {
        log.clear();
    }
    std::string print() {
        return log;
    }

    REGISTER_RPC_FUNCTIONS(Bar, append, clear, print);

    DEFAULT_SERIALIZATION_SUPPORT(Bar, log);
    Bar(const std::string& s = "") : log(s) {}
};
```

1.2 Executing operations on replicated state

The operations are executed by calls to the member function of the class representing the shard. The function must be executed at all members and in the same order. This is achieved by making an RPC call from user code at one of the members. An RPC or a remote procedure call is the mechanism by which one executes a function at a remote server. The client of the RPC needs to know at least the function identifier (name) and the arguments. The client *serializes* function name and arguments (also called *marshaling*) to a string of bytes in order to send them over the network to the remote server. The server after receiving the bytes, *deserializes* the string to retrieve the function name and the list of arguments (also called *demarshaling*). Below we see code examples of how the RPC calls are made:

```
Replicated<Foo>& foo_rpc_handle = group->get_subgroup<Foo>();
Replicated<Bar>& bar_rpc_handle = group->get_subgroup<Bar>();
int new_value = 3;
cout << "Changing Foo's state to " << new_value << endl;
derecho::rpc::QueryResults<bool> results = foo_rpc_handle.
  ordered_query<RPC_NAME(change_state)>(new_value);
decltype(results)::ReplyMap& replies = results.get();
cout << "Got a reply map!" << endl;
for(auto& reply_pair : replies) {
  cout << "Reply from node " << reply_pair.first << " was " <<
  std::boolalpha << reply_pair.second.get() << endl;
}
cout << "Appending to Bar" << endl;
bar_rpc_handle.ordered_send<RPC_NAME(append)>("Write from 1...");
```

The node first obtains the handle of the subgroup from the Derecho group by calling *get_subgroup*. The template argument specifies the type (first line has Foo, second has Bar). There's an optional argument (0 by default) to get_subgroup which provides the index of the desired subgroup of that type since multiple subgroups of a given type are supported. The resulting handle is the replicated object which supports the function ordered_query (or similarly, ordered_send if no return value is expected). The template argument of ordered_query is the function tag which specifies the function. This function must have been registered with Derecho when the class was defined (refer back to the REGISTER_RPC_FUNCTIONS macro). The argument to ordered_query are the function arguments. Upon successful execution of the ordered_query function, Derecho regusters the request and returns a QueryResults<return_type>object which the client can use in the future to query for the results of the object. This is an example of asynchronous programming. Derecho will send the RPC request from the local node as a message to the subgroup which when delivered will result in the execution of the function at each member. Each one of them will return the result back to the requesting node and the user can query for the result using this QueryResults object. At the time of making the request, you aren't sure which nodes are going to reply to you since a view change can trigger change of membership. By calling *results.get()*, the client knows of the members that will be replying (this is a blocking call, but non-blocking versions of querying exist and can be taken advantage of, as per the application requirements). This is captured by the replies object. Then iterating through replies, the client can wait for reply from each member (reply_pair.second.get() in the code). Similarly, the last line shows an example of an ordered_send which appends "Write from 1..." to the log of the Bar subgroup. It does not return anything.

1.3 Persistence and versioning

By default, each object is maitained in memory. But, for better fault tolerance, persisting data on disk is the way to go. One cannot recover state in volatile memory if all members of a subgroup fail simultaneously, so if the data is persisted in disk, the subgroup can restart with different nodes that first reload the log from disk to construct the state in memory. Versioning is useful in a similar way. Each state change can be seen as making a new version of the replicated state. Keeping all versions starting from the initial state or just maintaining a few recent versions either in memory or on disk (called version-vector) helps us query data efficiently. While a state change must involve every replica and thus go through the atomic multicast protocol, a query (or read) can work with past versions of the data provided we do that consistently. If so, a query can only visit only one replica and be more efficient as a result. Derecho provides temporally-precise and causally-consistent version-vector querying. Versions can be indexed either by position (version number 5 for example) or time. Thus, query every shard of the system (one member of each shard) for a version at 10 AM in the morning will give me a snapshot of the sytem that is causally consistent (if I see the result of an action at 10 AM, the action was captured before 10 AM and factors in the result).

In the code below, you see an example of a verion vector in volatile memory:

```
/**
* Non-Persitent Object with vairable sizes
*/
class ByteArrayObject: public mutils::ByteRepresentable {
  public:
  Persistent<Bytes,ST_MEM> vola_bytes;

  void change_vola_bytes(const Bytes& bytes) {
    *vola_bytes = bytes;
  }

  Bytes query_vola_bytes(uint64_t query_us) {
    HLC hlc{query_us,0};
    try{
      return *vola_bytes.get(hlc);
    } catch (std::exception e){
      std::cout<<"query_vola_bytes failed:"<<e.what()<<std::endl;
    }
    return Bytes();
  }

```

```
    REGISTER_RPC_FUNCTIONS(ByteArrayObject, query_vola_bytes,
    query_const_int, query_const_bytes, change_vola_bytes);

    DEFAULT_SERIALIZATION_SUPPORT(ByteArrayObject,vola_bytes);
    // constructor
    ByteArrayObject(Persistent<Bytes,ST_MEM> & _v_bytes):
    vola_bytes(std::move(_v_bytes)) {
    }
    // the default constructor
    ByteArrayObject(PersistentRegistry *pr):
    vola_bytes(nullptr,pr) {
    }
};
```

The vola_bytes object of the class is a Persistent object of type Bytes, persisteted in memory (ST_MEM). The function *query_vola_bytes* indexes by time *query_us*. The basic function available to the user is Persistent<T>::get which is overloaded to provide indexing by sequence number as well as time.

```
uint64_t query_ts_us = center_ts_us + random();
auto shard_iterator = group->get_shard_iterator<ByteArrayObject
  >();
auto query_results_vec = shard_iterator.p2p_query<RPC_NAME(
  query_vola_bytes)>(query_ts_us);
for(auto& query_result:query_results_vec) {
  auto& reply_map = query_result.get();
  PayLoad *pl = (PayLoad*)reply_map.begin()->second.get().bytes;
  volatile uint32_t seq = pl->msg_seqno;
  seq = seq;
}
```

The shard_iterator object iterates through each shard and runs the p2p_query that calls query_vola_bytes. A P2P query is different from an ordered query in that it makes a request to one other node of the subgroup as opposed to the entire subgroup.

2 Data-plane control-plane separation and SST programming

In many applications, including Derecho, there's movement of data followed by coordination among nodes as dictated by application logic. This coordination is driven by data and may involve message receive acknowledgements, failure suspicions etc. In turn, it drives the data plane forward (for example, a message acknowledgement can mean that you send the next message). Derecho uses RDMC for data transfer and SST for state management (refer to the previous recitation notes). The two most important control plane operations for Derecho are determining message delivery (when is it safe to deliver a message in a way that every non-failed member will also deliver it in the same order) and membership management (track who has failed and propagate failure suspicions through SST. Coordinate to decide on the next view membership). It is always a good idea to separate your concerns in systems design. The data plane of the system generally involves sending large amounts of data. Thus a primary performance requirement is high throughput. The control plane requires exchaning small important messages that therefore, demand low latency. RDMC and SST have been designed with these goals in mind. To see what it means to "separate" the two layers, we first see how they can interfere with each other. A naïve scheme can send one message, then wait for coordination and then send another. This is inefficient because the coordination (involving small messagaes) is not going to use all of the network bandwidth. Plus, if a node is slow in sending control messages, nodes are going to wait doing nothing. A better scheme will be to not have the two layers depend on each other for performance (of course, they rely on each other to move the system forward). Derecho achieves this by sending new messages and coordinating on past messages at the same time (thus masking coordination overhead, in effect). By maintaining a big enough window of messages that have not been delivered, and smart coordination protocols that work in batches, it is possible to achieve a case of a fully asynchronous pipeline of message send and delivery in which the nodes never wait to send data. Another advantage of batched coordination is for slow nodes. For example, a slow node can catch up and acknowledge all the messages it received since the last acknowledgement. SST makes it possible using what we call monotonic logic. Values in SST columns grow monotonically and protocols work no matter when a node reads those values in its memory (which are changing constantly since other nodes are writing to the local copy). Another way to see that is a node's knowledge of the system state grows monotonically and therefore, conclusions dervied from that knowledge continue to be true in the future. Protocols handle this "expansion" of knowledge in a way that allows a node to batch multiple events together. For concrete examples, see the lecture slides.