Allowing multiple threads with a global resources

Irfan Pyarali and Michael Kircher

October 11, 1999

1 Introduction

2 Overview

2.1 Reactor

Restrictions:
Only one thread is allowed to call handle_events!

2.2 Leader - follower model

Splitting threads into leaders and one follower. The leader is in possession of the reactor. It will be the only one getting callbacks from the reactor because of that. Therefore it has to dispatch these events to the follower - to the right follower.

3 Idea

3.1 Rules

- A Client_Connection_Handler is attached to a specific thread.
- If the leader gets its response, new leder has to chosen, if available
- If the leader arrives again in the Client_Connection_Handler.send_request nested upcalls are done.
- When a follower Client_Connection_Handler becomes active again, it has to check if the response is available or if it has to become the leader.
- If the leader gets a response, which is not its response it has to signal the corresponding follower.

- The thread has to check in the Client_Connection_Handler callback method handle_input, if it is the thread the Client_Connection_Handler expects. If not is has give control to the Client_Connection_Handler's thread.
- As a leader suspend the follower from the reactor if its response arrrived.
- The Client_Connection_Handler has to be able to identify its thread.
- The Client_Connection_Handler has to be able to wait on a condition variable, it is only used by itself and is signaled by the leader.
- The reactor does locking inside the handle_events, suspend_handler and resume_handler methods. handle_input cannot use the same lock to protect common variables as to protected the aforementioned three methods.

3.2 Invariants

- A Client_Connection_Handler waiting on a the condition variable can be accessed by exactly one other thread, the leader.
- A Client_Connection_Handler calling ACE_Reactor.handle_events cannot be in the follower queue.
- Only the leader is allowed to call ACE_Reactor.handle_events.

4 Specification

4.1 Additions

introduce in the ORB:

```
Unbounded_Set follower_set;
Lock leader_follower_lock;
boolean leader_available;
Thread_t leader_thread_ID;
```

Client_Connection_Handler:
introduce:

```
Thread_t thread_ID;
boolean update_to_leader;
// true if needed to update to a leader,
// false meaning the reponse is available
condition_variable cond_response_available;
```

already exisiting:

```
boolean input_available;
boolean expecting_response;
```

4.2 Client_Connection_Handler.send_request

```
this.expecting_response = true;
this.thread_ID = self;
lock (ORB.leader_follower_lock);
if (ORB.leader_available == true) and (ORB.leader_tread_ID != self) then
  ORB.follower_set.add (this);
  cond_wait (this.cond_response_available, ORB.leader_follower_lock);
  unlock (ORB.leader_follower_lock);

  if (this.update_to_leader == true) then
    // become the leader
    while (!this.input_available) ACE_Reactor.handle_events ();
  else
    // there is input waiting for me
    result this.handle_input (); // fake the handle_input
    if (result == -1)
      ACE_Reactor.remove_handler (this);
      // failure handling
      return;
    else
      // result == 1
      // we have to reschedule, not implemented yet
    endif
  endif
else if (ORB.leader_available == true) and (ORB.leader_thread_ID == self) then
  // we are already the leader, so we do nested upcalls
  unlock (ORB.leader_follower_lock);
  while (!this.input_available) ACE_Reactor.handle_events ();
else
  // become a leader, because there is no leader (also no follower)
  ORB.leader_available = true;
  ORB.leader_thread_ID = myself;
  unlock (ORB.leader_follower_lock);
  while (!this.input_available) ACE_Reactor.handle_events ();
endif;

// go on as usually
this.input_available = false;
ACE_Reactor.resume_handler (this);
this.expecting_response = false
```

4.3 Callback routinge Client_Connection_Handler.handle_input

```
int retval = 0;


ORB.leader_follower_lock.acquire ();

if (!this.expecting_response_)
{
  // we got something, but did not want
  // @@ wake up an other thread, we are lost

  // We're a client, so we're not expecting to see input.  Still
  // we better check what it is!
  ssize_t ret;
  ACE_Time_Value tv = ACE_Time_Value::zero;
  retval = 0;

  retval = -1;
  if (this->calling_thread_ == 0)
  {
    ret = this.peer.recv (MSG_PEEK, &tv);
    retval = 0;
    // if -1 is returned, the nested upcalls server crashes,
    // if the tv value is not specified we will hang in a blocking read
  }
  else if (ret = this.peer.recv (MSG_PEEK) > 0)
  {
    ret = this.peer.recv_n ();
  }

  // We're not expecting input at this time, so we'll always
  // return -1 for now.
  // -1 and rec_n with tv = 0 worked

  // error handling of ret

  TAO_ORB_Core_instance ()->leader_follower_lock ().release ();
  return retval;
}

if (this.calling_thread_ == self ())
{
  // we are now a leader getting its response
  // or a follower faking the handle_input

  this.input_available_ = 1;
```

```
  ORB.leader_follower_lock.release ();

  ORB.reactor.suspend_handler (this);

  return 0;
}
else
{
  // we are a leader, which got a response for one of the followers,
  // which means we are now a thread running the wrong Client_Connection_Handler

  ORB.remove_follower (this.cond_response_available_);
  ORB.leader_follower_lock.release ();

  ORB.reactor.suspend_handler (this);
  ORB.leader_follower_lock.acquire ();

  // the thread was already selected to become a leader,
  // so we will be called again.
  this.input_available_ = 1;
  this.cond_response_available_.signal ();

  ORB.leader_follower_lock.release ();
  return 0;
```

5 Implementation

5.1 TAO_ORB_Core

```
..
public:
int
TAO_ORB_Core::leader_available (void)
// returns the value of the flag indicating if a leader
// is available in the leader-follower model
{
  return this->leaders_;
}

int
TAO_ORB_Core::I_am_the_leader_thread (void)
// returns 1 if we are the leader thread,
// else 0
{
  if (this->leaders_)
    return (this->leader_thread_ID_ == ACE_Thread::self ());
  else
    return 0;
}

void
TAO_ORB_Core::set_leader_thread (void)
// sets the thread ID of the leader thread in the leader-follower
// model
{
  ACE_ASSERT ((this->leaders_ >= 1 && this->leader_thread_ID_ == ACE_Thread::self ())
             || this->leaders_ == 0);
  this->leaders_++;
  this->leader_thread_ID_ = ACE_Thread::self ();
  ACE_DEBUG ((LM_DEBUG,
              "(%P|%t) TAO_ORB_Core: New leader: leaders = %d\n",
              this->leaders_));
}

int
TAO_ORB_Core::unset_leader_wake_up_follower (void)
// sets the leader_available flag to false and tries to wake up a follower
{
  ACE_Guard <ACE_SYNCH_MUTEX> g (TAO_ORB_Core_instance ()->leader_follower_lock ());

  ACE_DEBUG ((LM_DEBUG,
              "(%P|%t) TAO_ORB_Core: unset leader\n"));
  this->unset_leader_thread ();
```

```
  if (TAO_ORB_Core_instance ()->follower_available ()
    && !this->leader_available ())
  // do it only if a follower is available and no leader is available
  {
    ACE_SYNCH_CONDITION* condition_ptr = this->get_next_follower ();
    if (this->remove_follower (condition_ptr) == -1)
      return -1;
    condition_ptr->signal ();
  }
  else
    ACE_DEBUG ((LM_DEBUG,
                "(%P|%t) TAO_ORB_Core: no new leader: "
                "follower = %d leaders = %d  leader_ID = %d\n",
                TAO_ORB_Core_instance ()->follower_available (),
                this->leaders_,
                this->leader_thread_ID_));
  return 0;
}


void
TAO_ORB_Core::unset_leader_thread (void)
// sets the flag in the leader-follower model to false
{
  ACE_ASSERT ((this->leaders_ > 1 && this->leader_thread_ID_ == ACE_Thread::self ())
             || this->leaders_ == 1);
  this->leaders_--;
}


ACE_SYNCH_MUTEX &
TAO_ORB_Core::leader_follower_lock (void)
// returns the leader-follower lock
{
  return this->leader_follower_lock_;
}

int
TAO_ORB_Core::add_follower (ACE_SYNCH_CONDITION *follower_ptr)
// adds the a follower to the set of followers in the leader-
// follower model
// returns 0 on success, -1 on failure
{
  ACE_DEBUG ((LM_DEBUG,
             "(%P|%t) TAO_ORB_Core::add_follower: add \n"));
```

```
  if (this->follower_set_.insert (follower_ptr) != 0)
    return -1;
  return 0;
}

int
TAO_ORB_Core::follower_available (void)
// checks for the availablity of a follower
// returns 1 on available, 0 else
{
  return !this->follower_set_.is_empty ();
}

int
TAO_ORB_Core::remove_follower (ACE_SYNCH_CONDITION *follower_ptr)
// removes a follower from the leader-follower set
// returns 0 on success, -1 on failure
{
  return this->follower_set_.remove (follower_ptr);
}

ACE_SYNCH_CONDITION*
TAO_ORB_Core::get_next_follower (void)
// returns randomly a follower from the leader-follower set
// returns follower on success, else 0
{
  ACE_Unbounded_Set_Iterator<ACE_SYNCH_CONDITION *> iterator (this->follower_set_);
  if (iterator.first () == 0)
    // means set is empty
    return 0;
  return *iterator;
}

private:
  static ACE_SYNCH_MUTEX leader_follower_lock_;
  // do protect the access to the following three members

  static ACE_Unbounded_Set<ACE_SYNCH_CONDITION *> follower_set_;
  // keep a set of followers around (protected)

  static int leaders_;
  // 0 if no leader is around, 1 if there is a leader
  // > 1 if we do nested upcalls (protected)

  static ACE_thread_t leader_thread_ID_;
  // thread ID of the leader thread (protected)
```

5.2 TAO_Client_Connection_Handler

5.2.1 Declarations

introduce:

```
private:
  ACE_thread_t calling_thread_;
  // the thread ID of the thread we were running in.

  ACE_SYNCH_CONDITION* cond_response_available_;
  // wait on reponse if the leader-follower model is active
```

already exisiting:

```
private
  int expecting_response_;
  // State flag which, if non-zero, indicates that this handler is
  // looking to get input.  Otherwise, any input received is
  // unexpected.

  int input_available_;
  // Flag indicating whether or not input is available.  Only valid
  // when <expecting_response_> is non-zero.
```

5.2.2 TAO_Client_Connection_Handler::send_request

5.2.3 TAO_Client_Connection_Handler::handle_input

5.3 default_client and TAO_Client_Strategy_Factory

Declaration:

```
  ...
public:
  ACE_Lock* create_leader_follower_lock (void);
  // create the lock for leader follower model used
  // with global resources and several threads running.
private:
  Lock_Type iiop_leader_follower_lock_type_;
  // the lock type for the leader follower model
  ...
```

default_client::parse_args ():

```
...
if (ACE_OS::strcmp (argv[curarg], "-ORBleaderfollowerlock") == 0)
  {
    curarg++;
    if (curarg < argc)
      {
        char *name = argv[curarg];

        if (ACE_OS::strcasecmp (name, "thread") == 0)
          this->leader_follower_lock_type_ = TAO_THREAD_LOCK;
        else if (ACE_OS::strcasecmp (name, "null") == 0)
          this->iiop_profile_leader_follower_type_ = TAO_NULL_LOCK;
      }
  }
...
```

default_client::create_leader_follower_lock ():

```
ACE_Lock *
TAO_Default_Client_Strategy_Factory::create_leader_follower_lock (void)
{
  ACE_Lock* the_lock = 0;

  if (this->leader_follower_lock_type_ == TAO_NULL_LOCK)
    ACE_NEW_RETURN (the_lock,
    ACE_Lock_Adapter<ACE_SYNCH_NULL_MUTEX> (),
    0);
  else
    ACE_NEW_RETURN (the_lock,
    ACE_Lock_Adapter<ACE_SYNCH_MUTEX> (),
    0);

  return the_lock;
}
```