Implementation of Resilient, Atomic Data Types

William Weihl and Barbara Liskov
Presented by: Sheng-Yih Wang

September 18, 1996

Introduction

- Preserve consistency in the presence of concurrency and hardware failures
- Atomic actions (Transactions)
  - Serializability
  - Recoverability — effect is all-or-nothing
- Atomic objects, atomic types
- Resilient data

Atomic types

- Provide serializability and recoverability
- Safety requirement and Liveness requirement
- Atomicity of action — global property
- Atomicity for a type — local property
- Local atomicity — dynamic atomicity or static atomicity

- Argus uses dynamic atomicity
- Dynamic atomicity — defines limits on the concurrency permitted
- Conflict of two actions — if one has observed the effect of the other, or if one has invalidated the result if the other's operations.
- Implementation of dynamic atomicity — two-phase locking
- Two-phase locking $\subset$ dynamic atomicity
- Dynamic atomicity allows types to have both partial and nondeterministic operations.

Model of Computation (Argus)

- Node — an independent computer
- Nodes communicate only by sending messages
- In Argus, an application is implemented from one or more *guardians*
- Guardian
  - consist of some data objects and processes
  - provide access of its objects via *handlers*
  - each guardian resides at a single physical node
  - guardian's *stable object* survives crashes
  - runs a special recovery process when recovering
  - copy the *stable object* to stable storage periodically
- Argus provides build-in atomic actions and atomic data types
- Actions are the primary methods of carrying out computations
- Serializability is implemented by strict two-phase locking
- Recoverability is implemented by modifying a copy of the object

User-defined Atomic Types

- Mutex
  - action concurrency and process concurrency
  - provide mutual exclusion
  - operations: create, get_value, seize, pause
  - create = **proc**(x: T) **returns**(**mutex** [T])
  - get_value = **proc**(m: **mutex** [T]) **returns**(T)
  - m.value = get_value(m)
  - **seize** *expr* **do** *body* **end**
- Semiqueue
  - operations: create, enq, deq (remove arbitrary elements)
  - Liveness constraint: deq must eventually remove every eligible element
  - **rep** = atomic_array[elem]
  - enq = addh, deq = addl

Resilience

- Mutex (extended)
  - synchronization of user process
  - notifying the system to write to stable storage
  - defining what information is written to stable storage
  - ensuring that information written is consistent
- **changed**: mutex operation — write to stable storage
- Incrementally copy — don't copy contained mutex and atomic objects
- Example: Amap

Semique

```
semiqueue = cluster is create, enq, deq
qitem = atomic_variant[present: int, absent: null]
buffer = array[qitem]
rep = mutex[buffer]

create = proc() returns (cvt)
  return(rep$create(buffer$new()))
end create

enq = proc(q: cvt, i: int)
  item: qitem := qitem$make_absent(nil) % “absent” if action aborts
  qitem$change_present(item, i) % “present” if action commits
  seize q do
    buffer$addh(q.value, item) % add new item to buffer
  end
  rep$changed(q) % notify system of modification to buffer
end enq
```

```
deq = proc(q: cvt) returns(int)
   cleanup(q)
   seize q do
      while true do
         % look at all items in the buffer
         for item: qitem in buffer$elements(q.value) do
            tagset item % see if item can be dequeued by this action
               wtag present(i: int): qitem$change_absent(item, nil)
                        return(i)
            end % tagset
         end % for
         pause
      end % while
   end % seize
end % deq

cleanup = proc(q: rep)
   enter topaction % start an independent action
      seize q do
         b: buffer := q.value
         for item: qitem in buffer$elements(b) do
            tagset item
               tag absent: buffer$reml(b) % remove only qitems in the “absent” sta
               others: return
            end % tagset
         end % for
      end % seize
   end % enter — commit cleanup action here
end % cleanup

end semiqueue
```

Amap

```
amap = cluster[vtype: type] is create, insert, delete, lookup
status = atomic_variant[present: vtype, absent: null]
table = map[vtype]
log = map[status]
rep = record[items: mutex[table], intentions: mutex[log]]

create = proc() returns (cvt)
  return(rep${intentions: mutex[log]$create(log$create()),
      items: mutex[table]$create(table$create())})
end create

insert = proc(m: cvt, u: uid, v: vtype) signals(duplicate)
  seize m.intentions do
    while true do
      s: status := find_status(m,u)
      tagset s
        tag present: signal duplicate
        wtag absent: status$change_present(s, v)
          mutex[log]$changed(m.intentions)
          return
        end % tagset
        pause % couldn't lock s; wait and try again
      end % while
  end % seize of intentions
end % insert
```

```
delete = proc(m: cvt, u: uid) signals(not_found)
   seize m.intentions do
      while true do
         s: status := find_status(m,u)
         tagset s
            wtag present: status$change_absent(s, nil)
               mutex[log]$changed(m.intentions)
               return
            tag absent: signal not_found
            end % tagset
         pause % couldn't lock s; wait and try again
         end % while
   end % seize of intentions
end % delete

lookup = proc(m: cvt, u: uid) return(vtype) signals(not_found)
   seize m.intentions do
      while true do
         tagset find_status(m,u)
            tag present(v: vtype): return(v)
            tag absent: signal not_found
            end % tagset
            pause % couldn't lock status object; wait and try again
         end % while
   end % seize of intentions
end % lookup
```

```
find_status = proc(m: rep, u: uid) returns(status)
   % m.intentions has been seized in caller
   l: log := m.intentions.value
   if log$size(l) > 1000
      then rebuild(m) end % garbage collection
   return(log$lookup(l,u))
      except when not_found: end
   s: status
   seize m.items do
      s:= status$make_present(table$lookup(m.items.value, u))
      end except when not_found: s:= status$make_absent(s, nil) end
   log$insert(i, u, s)
   return(s)
end find_status
```

```
rebuild = proc(m: rep)
   % m.intentions has been seized in caller
   l: log := m.intentions.value
   enter topaction % start a new "rebuild" action
      seize m.items do
         t: table := m.items.value
         for u: uid, s: status in log$pairs(l) do
            tagset s
               wtag present(v: vtype):
                  log$delete(l, u)
                  table$alter(t, u, v)
                  mutex[table]$changed(m.items)
               wtag absent:
                  log$delete(l, u)
                  table$delete(t, u)
                  mutex[table]$changed(m.items)
               end % tagset
                  except when unchanged, not_found: end
         end % for
      end % seize
   end % enter — rebuild action commits here
end % rebuild
end amap
```