On the multiple facets of call synchronization

Runtimes for concurrency and distribution

Tullio Vardanega, tullio.vardanega@unipd.it

Academic year 2024/2025

Evaluating language features – 1/2

- The features of a programming language may be evaluated from two orthogonal angles
- **Expressive power**
  - How far they go about addressing the user needs
  - Letting the programmer express their intended meaning
- **Usability**
  - How well individual language features do on their own (*efficacy*)
  - Versus how well they interact with each other (*coherence*)
  - Feature interaction is a very pain point of language design

Evaluating language features – 2/2

- The synchronization constructs are especially relevant to an evaluation of that kind

  T. Bloom (1979)
  Evaluating synchronization mechanisms
  DOI: 10.1145/800215.806566

- Bloom's study singled out six distinct types of conditions that can be used to specify conditions on synchronization
  - Over and above exclusion synchronization

Conditions on synchronization – 1/2

1. **Contingent on the *synchronization state* of the resource** ☑
   - Analogous to the case of the "guardian" that we have already encountered: applying service rules on sets of enqueued calls
   - Use of the `'Count` attribute tells when to open or close guards
2. **Contingent on the *logical state* of the resource** ☑
   - The classic case of "no-write-on-full", "no-read-on-empty"
   - Guards capture those application-specific conditions
3. **Contingent on the *history of service*** ☑
   - When fairness, load balance, energy efficiency matter

Conditions on synchronization – 2/2

4. **Contingent on the *type of request*** ☑
   - Preferential treatment for some requests over others, including types of callers
   - Analogy: in certain SSDs, alternating individual read/write operations is less efficient than serving them in groups
5. **Contingent on the *time of the request*** ☑
   - For example reflected in FIFO queuing
6. **Contingent on the *request parameters*** ☒
   - Where serviceability depends on whether the server can dispense as much as requested
     - E.g., as in paging or heap management

The resource allocation problem – 1/3

- Recurrent problem in any concurrent system
  - It involves all of Toby Bloom's 6 dimensions
- Our current model is *unable* to handle it properly
- **Example**

a) A resource manager dispenses a finite quantity $\boldsymbol{N}$ of resources $\{R_{j=1,\dots,N}\}$

b) Concurrent clients $\{C_{i=1,\dots,M}\}$ may request any amount $\boldsymbol{Q}_i \le \boldsymbol{N}$ of those resources

c) Accepted requests shall be satisfied fully

The *wait-until-satisfied* (and you will be) service protocol is most precious

With it, requests *cannot* return until satisfied

d) Clients return resources after use

The resource allocation problem – 2/3

- What does the problem specification tell us?
- Client interaction with resource manager must be *synchronous*
  - Wait-until-satisfied (requirement c)
- And must be *predictable* : service shall be rendered at some point
- The volume of request is specified as a request parameter
  - This is the only sensible interface of the server
- The quantity can only be known *after* synchronization has started
- What happens when the server finds itself unable to satisfy the request being examined?
- It *cannot* return to the caller prematurely (requirement c)
  - Hence it must keep that request on hold
- How can it do that while continuing to serve others?
  - Serving others is essential because it allows releases (requirement d)

The resource allocation problem – 3/3

- Do guards help in this case?
  - *No, they don't* !
- They prevent synchronization when the requested service cannot be executed
- But guards operate **before** client-server synchronization takes effect, hence **before** the request parameters can be examined
- Two alternatives are possible, both of which would need enhanced capabilities

1. Allowing guard expressions to access request parameters *without* engaging synchronization
2. Transferring to another queue the request that cannot be satisfied immediately
   - Beginning service but then suspending (parking) it until further notice
   - Becoming able to serve other pending requests

Alternative 1

$n = 1$ resource per request (trivial)

```
protected Controller is
 entry Allocate (R : out Resource);
 procedure Release (R : Resource);
private
 Free : Natural := Full_Capacity;
 …
end Controller;
protected body Controller is
 entry Allocate (R : out Resource)
  when Free > 0 is
 begin
  Free := Free - 1;
  …
 end Allocate;
 procedure Release (R : Resource) is
 begin
  Free := Free + 1;
 end Release;
end Controller;
```

$1 < n \leq N$ resources per request (much harder)

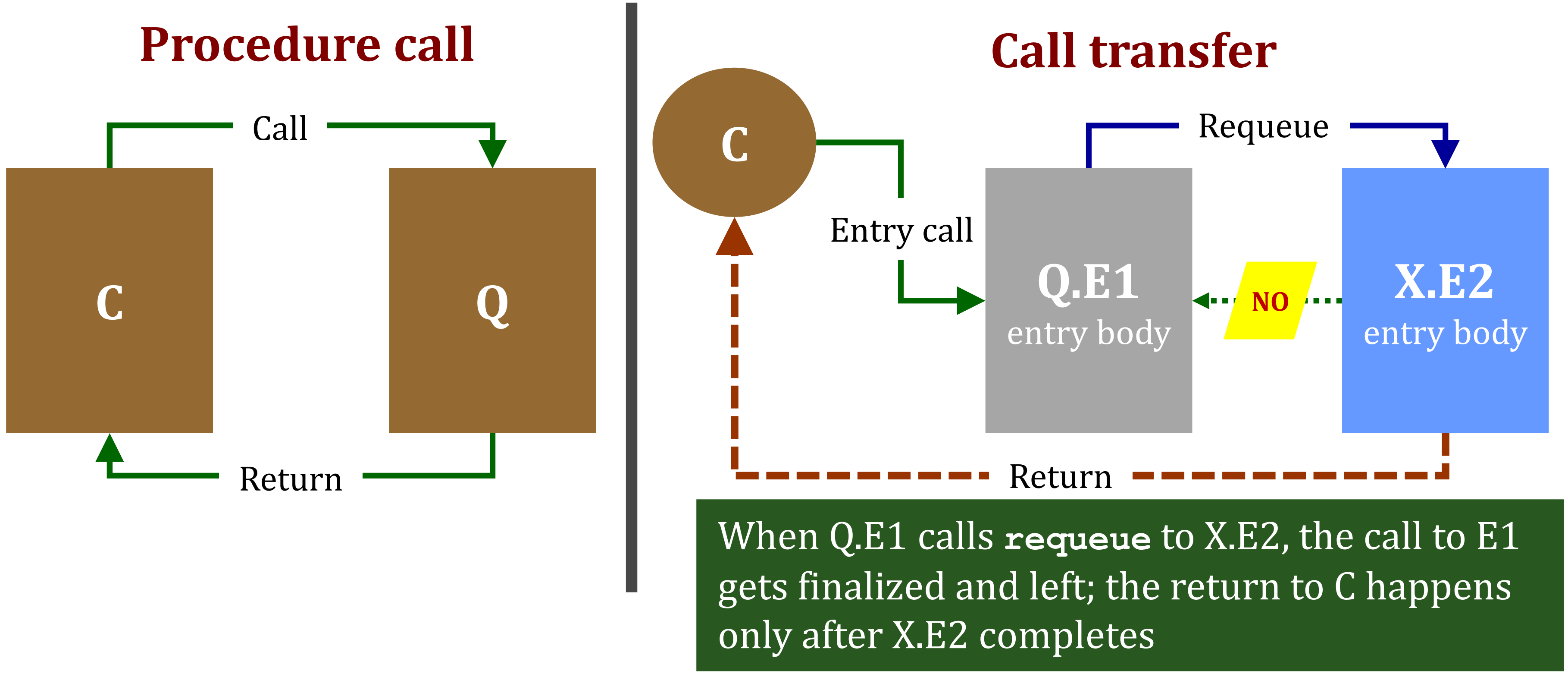
```
type Request is range 1..Max_Requests;
protected Controller is
 entry Allocate
  (R      :     out Resource;
   Amount : in      Request);
 procedure Release
  (R      : Resource;
   Amount : Request);
private
 Free : Request := Request'Last; …
end Controller;
protected body Controller is
 entry Allocate
  (R      :     out Resource;
   Amount : in      Request)
  when Amount <= Free is
 begin
  Free := Free – Amount;
 end Allocate;
 procedure Release (…) is …
end Controller;
```

Critique of alternative 1

- Requests that fail the guard are enqueued in the corresponding event queue (aka entry queue)
- Applying the eggshell model to retrieve serviceable calls would require traversing *the entire event queue* every time a R/W access to the server state completes
  - To seek any enqueued call for which the guard has become open
  - Which would be untenably costly in the general case
- Alternative 1 causes each individual call to have its own "state-change event"
  - Conditions would be *per-call*
- But the entry queue model that we know caters for a *single-condition* queue only
  - Conditions are *per-entry*

Alternative 2 – 1/5

- Transferring the call to another queue (**`requeue`**) is *not* a normal procedure call



Alternative 2 – 2/5

- Sophisticated feature, with challenging semantics
- Transferring the call to another queue should *neither* suspend the server on a closed guard at destination !
  - Doing so might cause deadlock
- *Nor* should it awake the client during the transfer
  - The service policy should be fully transparent to the caller
- Transfer should occur atomically, *without* undergoing guard evaluation at the target queue
- This raises two "feature-interaction" issues

1. Which entry queues can be allowable targets
2. What happens to a time-out set on the call

Alternative 2 – 3/5

- **Issue 1: allowable targets**
- *Any* entry with a *compatible* interface, anywhere, even *outside* of the server as long as in scope of the server
- The entry interface at the target is compatible if it is
  - Either identical to the one of origin
  - Or with additional parameters, but all with default values
  - Or with no parameters at all
- In all these cases, the runtime would know how to present the call at the target stack

Alternative 2 – 4/5

- **Implications**
- Transferring to a queue in the same server fits the eggshell model semantics nicely
  - In addition to yielding good functional cohesion
- Transferring to an entry queue outside of the server requires releasing the R/W lock held on it
- Without the eggshell model knowing exactly what to do at that point

Alternative 2 – 5/5

Issue 2: handling of time-out

- Two possible outcomes for client A

1) Call `B.E1` not accepted within `T1` gets cancelled
2) Call `B.E1` accepted and then transferred to `B.E2`, is cancelled if it does not get accepted *there* within `T1`

- Outcome 2) incurs an ugly temporal distortion, but its semantics makes sense from the client perspective
- **Important**: an accepted call is an *abort-deferred region*

Example

```
-- client A
select
 B.E1;
or
 delay T1;
end select;
```

```
-- server B
select
 accept E1 do
  … -- T2 time units

  requeue E2 with abort;
 end E1;
or
 …
end select;
```

The **`with abort`** clause preserves the time-out effect upon call transfer

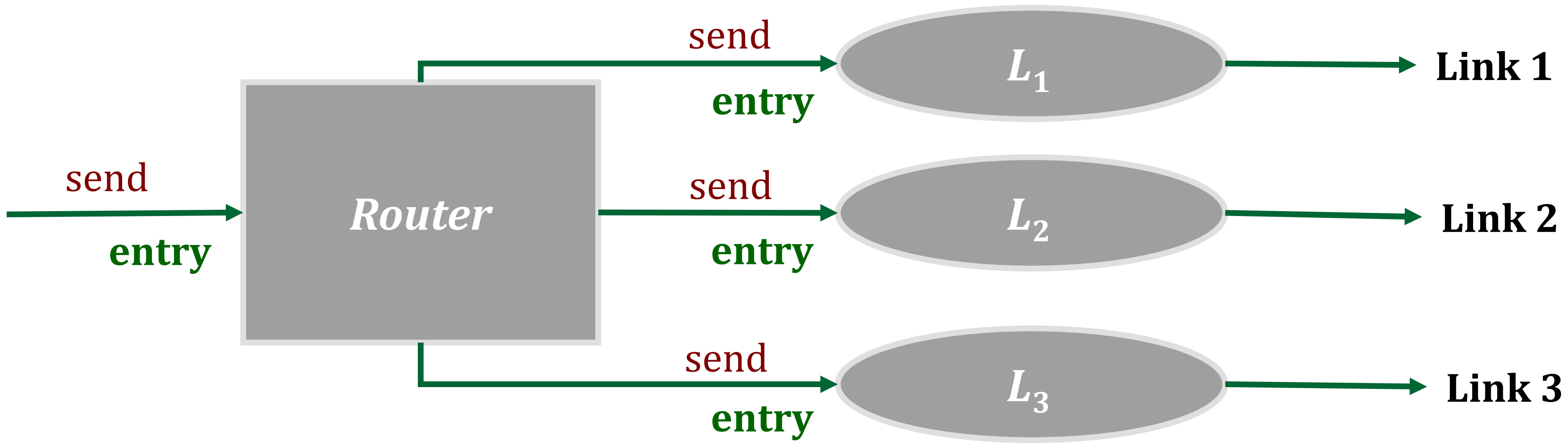
Use cases – 1/2

- Appropriate use of the `requeue` feature greatly facilitates the implementation of resource managers
  - Check the implementation linked to today's lecture: "programming example 3"
- **Homework**

1. Try and improve the given solution in a manner that avoids useless call transfers
2. Try the same solution with a programming language of your choice

Use cases – 2/2

- A network router may be able to forward inbound packets on to $N = 3$ outbound links $L_{i=1,\ldots,N}$
  - Link L_1 is the preferred choice, but the other links (first L_2 and then L_3) are used when L_1 risks overloading
- Likening packets to calls, and router and links to servers, maps packet forwarding to a requeue
- The service policy remains transparent to the router's client



In-class exercise

- Realize a circular-line metro service simulator
- $M > 1, M \in \mathbb{N}$ train stations along a circular line
- $N > M, N \in \mathbb{N}$ commuters who forever revolve around one and the same duty cycle
- Train with capacity $C < N$ (no prebooking)
- Commuter's duty cycle
  1. Go from home to nearest train station
  2. Board first possible train
  3. Get off the train and go to work (and work as due)
  4. Go from work to nearest train station
  5. Board first possible train
  6. Get off the train and go home (and rest as allowed)