Service Interface Design

Designing Service Interfaces

- In the software life-cycle, Interface Design is part of the **software design phase**:
  - Architecture definition: modules and their interfaces
  - Problems to be addressed in the design phase:
    - How to partition the system into modules, granularity
    - How to design interfaces
    - How to deploy modules onto hosts (distributed software)
- Software design is a creative activity. However, some fundamental principles have been identified
  - Basic Principle: **Information hiding**, i.e., clear separation between interface and implementation

Partitioning Principles

- Maximize internal cohesion
- Minimize coupling
- Foresee and favor future changes (openness to changes) and reuse (flexibility)
- In distributed environments, balance the load of the various distributed sub-systems

Choosing the Granularity Level

- The right trade-off must be found
- Too fine grain partitioning => negative impact on
  - performance
  - coupling
- Too coarse grain partitioning => negative impact on
  - internal cohesion
  - flexibility and reuse

Interface Design

- Must produce a formal description of interfaces
- Must be coherent with the partitioning principles
  - An interface must expose no more than what is strictly needed by the service client (information hiding)
  - An interface must have high internal cohesion and limit the number of interactions that are necessary to access its offered services (minimize coupling)
  - If an interface is object-oriented, inheritance should be adopted (favor re-use)

Interface Design

- Other best practices:
  - Foresee all particular cases (exceptions)
    - Favors debugging
    - Contributes to construct a robust and secure system
  - Prefer idempotent methods
    - Limits uncertainty on the results of distributed calls
  - Limit the number of interactions, but also the size of messages
    - Fragmentation, buffering and processing time may adversely affect performance
    - Allow clients to select what and how much to receive/send (i.e., avoid transmitting unnecessary data)

Interface Design Approaches

- **Method centric**
  - Fixed endpoint (classical programming language–like approach)
  - Design is about operations and their input/output arguments
- **Message centric**
  - Fixed, single-operation interface (e.g. send(Message))
  - Design is about messages and endpoints
- **Constrained**
  - Fixed, multiple-operation interface
  - Example: REpresentational State Transfer (REST)
  - Design is about endpoints, allocation of operations, and their input/output arguments

Half way between the previous two ones

Example: Bank Account Operations

- Service for enabling the execution of operations on a bank account
  - bank account identified by account id
  - operations:
    - read information about given account
    - deposit a given amount into given account
    - withdraw a given amount from given account
      - specify whether the requested amount can be reduced if not available in full
  - deposit and withdraw operations include a description (a string)
- Develop method-centric and constrained interface designs

Method-centric Design (1st try)

```
public interface AccountReader {
       public Account getAccountInfo(String id)
              throws UnknownAccountIdException;
}
```

Method-centric Design (1st try)

```
public interface AccountUpdater {
       public void addDeposit(
                      String id,
                      float amount,
                      String description)
              throws UnknownAccountIdException;

       public float addWithdrawal(
                      String id,
                      float requestedAmount,
                      boolean reducible,
                      String description)
              throws UnknownAccountIdException,
                      NoAvailabilityException;
}
```

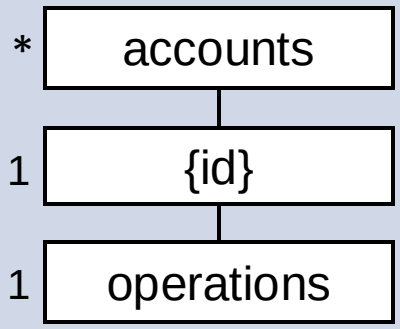
Method-centric Design (2nd try)

```
public interface ImprovedAccountUpdater {
       public void addDeposit(
                      String id,
                      float amount,
                      String description,
                      String transId)
              throws ReplicatedTransIdException,
                      UnknownAccountIdException,;

       public float addWithdrawal(
                      String id,
                      float requestedAmount,
                      boolean reducible,
                      String description,
                      String transId)
              throws ReplicatedTransIdException,
                      UnknownAccountIdException,
                      NoAvailabilityException;
}
```

REST Service Design (1st try)

Designing Resources

Resources	URLs	Repr.	Meaning
	accounts		set of all accounts
	accounts/{id}	account	single account
	accounts/{id}/operations		set of all operations of account identified by id

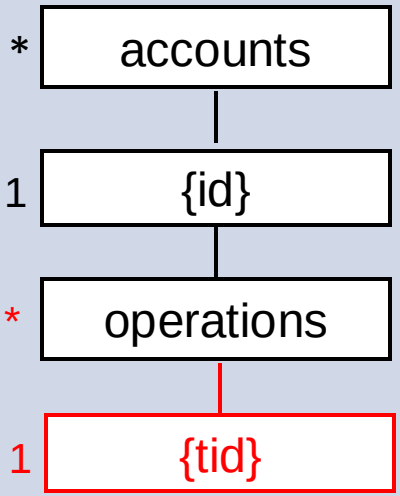
REST Service Design (1st try)

Designing Operations

Resource	Method	Req. body	status	Resp. body	meaning
accounts					
accounts/{id}	GET		200	account	OK
			404		Not found
accounts/{id}/operations	POST	operation	200	operation	OK
			409	reason	Conflict
			404		Not found

REST Service Design (2nd try)

Designing Resources

Resources	URLs	Repr	Meaning
	accounts		set of all accounts
	accounts/{id}	account	single account
	accounts/{id}/operations		set of all operations
	accounts/{id}/operations/{tid}	operation	single operation

REST Service Design (2nd try)

Designing Operations

Resource	Method	Req. body	status	Resp. body	meaning
accounts					
accounts/{id}	GET		200	account	OK
			404		Not found
accounts/{id}/operations /{tid}	PUT	operation	201	operation	Created
			409	reason	Conflict
			404		id Not found

Universality of Design Principles

- Design approach influences syntax, understandability
- However, most design principles apply to all design approaches because they are related to the underlying service features
- Example:
  - fault conditions may be represented as separate messages or as fields of a single message type or mapped to predefined error codes/messages.

  => In any case it is important to consider all kinds of faults in the design

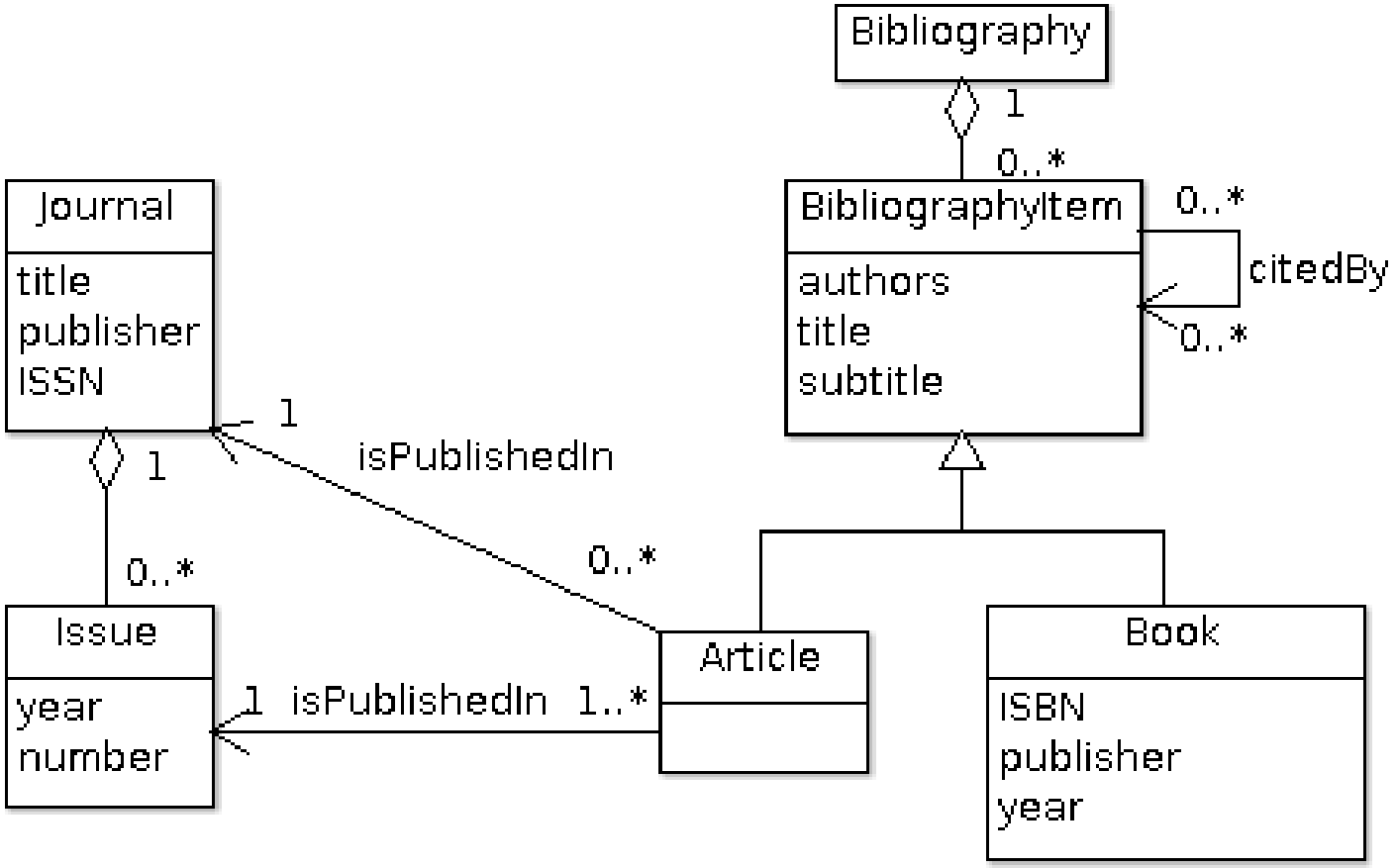
REST-specific Issues/Best Practices

- Designing a good RESTful API is not just a matter of
  - considering the general principles
  - adhering to the REST constraints
- There are other issues to be considered
  - how to design/organize resources
  - how to map conceptual operations to methods
  - how to provide documentation & API self-description capabilities
- There are design patters that can be followed
- Not every "REST API" is indeed fully REST

Running Example: Bibliography

- Design a RESTful web service for remote management of a bibliography:
  - Searching a bibliography
    - Search items by keyword, type (article/book) and publication year
    - For each item, get available data
    - Navigate through items by citation
  - Populating and updating a bibliography
    - Add new items to a bibliography
      - while adding, add also related data (e.g., journal, citations)
    - Modify or delete an item in a bibliography

Running Example: Object Model

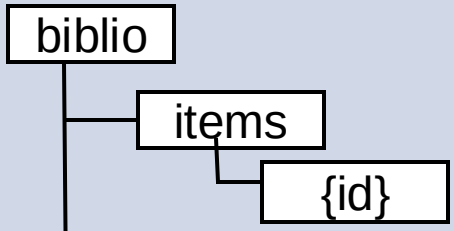
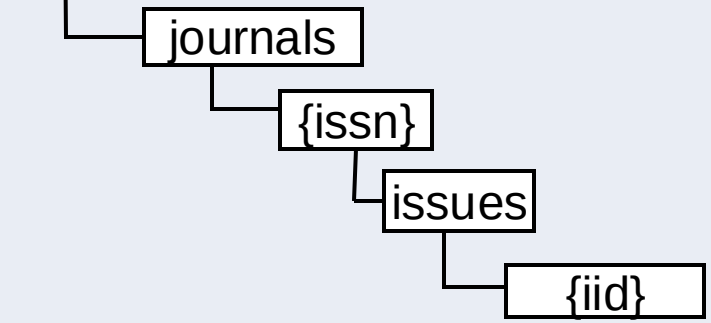


Designing Resources

- Start with a **conceptual** resource design (independent of URIs)
  - it should be possible to remap the resources to different URIs
  - resources correspond to single entities or collections of entities of the same type

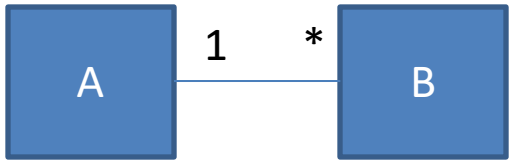
    => hierarchical organization (forest)
  - resources are unique but different resources can represent related or even overlapped underlying entities
    - Example: "the selected course" "the DSP course"
- Do not confuse resources (nouns) with actions (verbs)
- Use coarse granularity level (as far as possible)

Designing Resources for Biblio

Resources	Meaning
	The bibliography (main resource) The items in the bibliography The item uniquely identified by {id}
	The journals in the bibliography The journal with ISSN {issn} The issues of the journal with ISSN {issn} The issue with id {iid} of the journal with ISSN {issn}

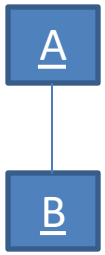
Representing Resource Relations

- Different mechanisms can be used:
  - references (hyperlinks)
  - resource nesting
- Example: 1 to many



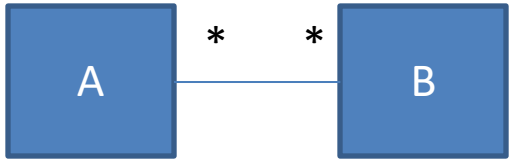
  - option 1:
    - B data model includes reference to A

  - option 2 (better when B not autonomous):
    - B instances (or references) nested into A instances

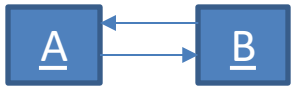


Representing Resource Relations

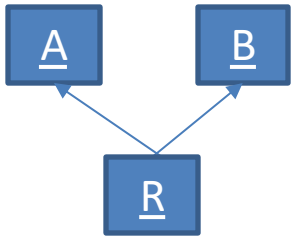
- Example: many to many



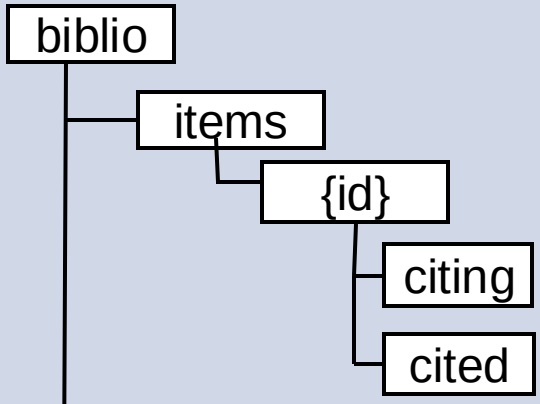
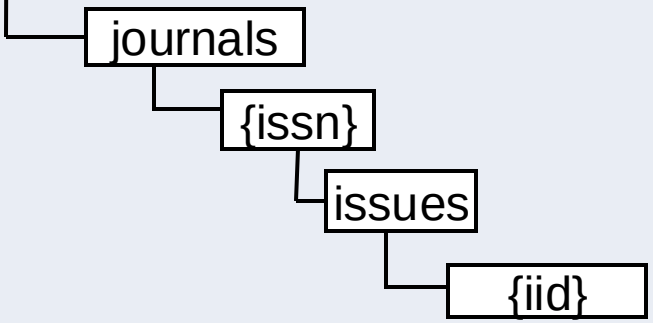
– option 1:
  - B data model includes references to A
  - A data model includes references to B



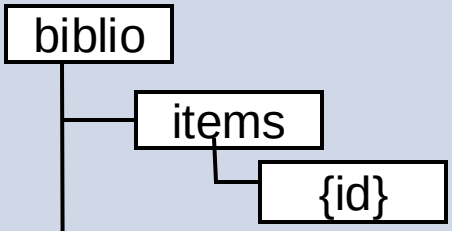
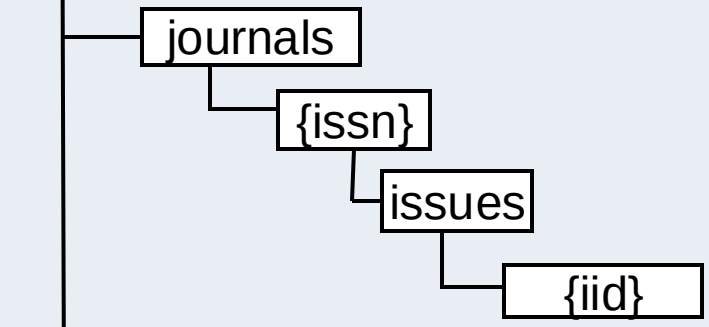
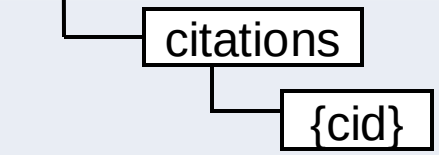
– option 2:
  - represent the elements of the relation, i.e. (A,B) pairs, as resources



Designing Resources for Biblio

Resources	Meaning
	The bibliography (main resource) The items in the bibliography The item uniquely identified by {id} The items that cite the item identified by {id} The items that the item identified by {id} cites
	The journals in the bibliography The journal with ISSN {issn} The issues of the journal with ISSN {issn} The issue with id {iid} of the journal with ISSN {issn]

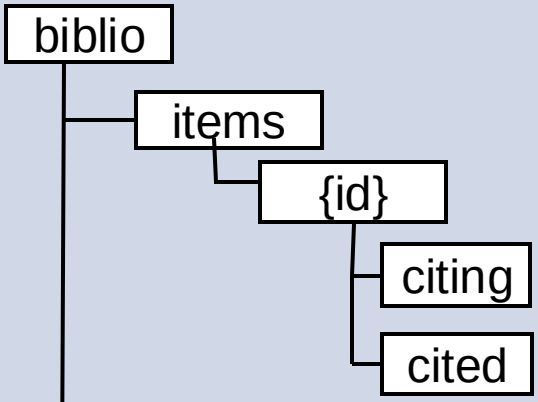
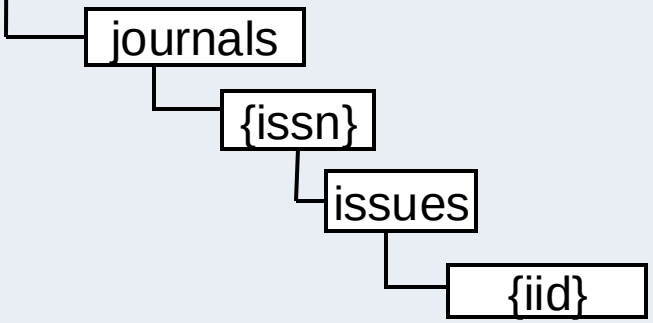
Designing Resources for Biblio (Alternative Solution)

Resources	Meaning
	The bibliography (main resource) The items in the bibliography The item uniquely identified by {id}
	The journals in the bibliography The journal with ISSN {issn} The issues of the journal with ISSN {issn} The issue with id {iid}
	the citations in the bibliography a single citation uniquely identified by {cid}

Mapping Resources to URIs

- Conventions contribute to making the API easy to understand and self-describing:
  - Use plural names for collections, singular for non-collections
  - Use URLs that reproduce the hierarchical relationships of resources
- However, according to HATEOAS, URLs should be opaque
- There is an open debate about this point

Mapping Resources for Biblio

Resources	Relative URLs
	biblio biblio/items biblio/items/{id} biblio/items/{id}/citing biblio/items/{id}/cited
	biblio/journals biblio/journals/{issn} biblio/journals/{issn}/issues biblio/journals/{issn}/issues/{iid}

Defining Resource Representations

- Define an **abstract data model** for each resource
  - in practice, the resource state
  - can be represented in various ways: UML, schema, ...
- Define a number of different **representations** of the resource data model
  - different content types

Representing Hyperlinks

- Data models may include references in the form of hyperlinks. Different conventions are in use
  - Link object (Atom)

```
"link" : {"rel:"come_from","href":"/country/italy"}
```

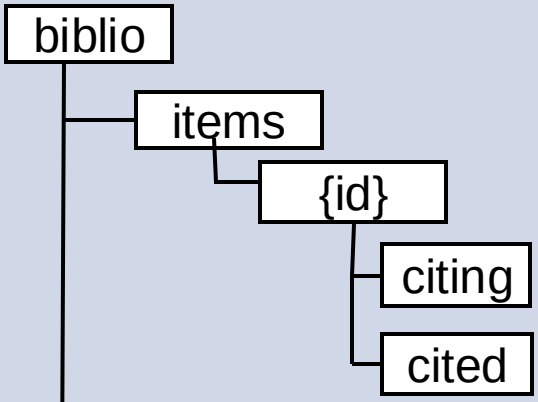
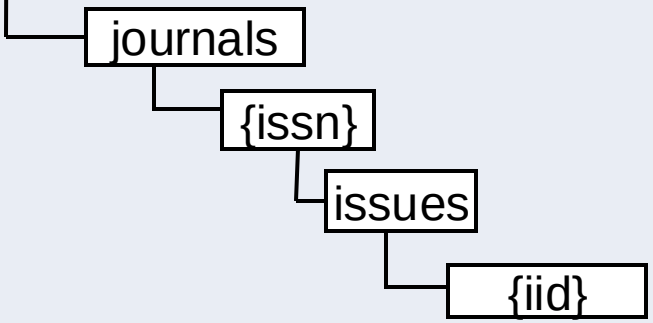
  - Object with href

```
"come_from" : {"href":"/country/italy"}
```

  - href string

```
"come_from" : "/country/italy"
```

Defining Resource Representations for Biblio

Resources	Representation (schema)
	biblio items item items items
	journals journal issues issue

Mapping Operations

- For each operation to be offered by the service
  - Find resource(s) and HTTP method(s) to be used
  - if no match, maybe some resource is missing?
- For each allowed resource/method pair, define
  - how the method has to be invoked (accepted query parameters, request headers and body contents)
  - the possible results (status codes) and, for each one of them, response headers and body contents

Mapping Operations for Biblio

- Search items by keyword, type (article/book) and publication year
- For each item, get available data

Resource	Verb	Query params	Status		Resp. Body
biblio/items	GET	keyword: string type: { "enum": ["article", "book"] } beforeInclusive:date afterInclusive:date	200	OK	filtered items (items)
biblio/items/{id}	GET		200	OK	item
			404	Not found	

Mapping Operations for Biblio

- Navigate through items by citation

Resource	Verb	Query params	Status		Resp. Body
biblio/items/{id}/citing	GET		200	OK	Items that cite {id} (items)
			404	Not Found	
biblio/items/{id}/cited	GET		200	OK	Items that {id} cites (items)
			404	Not Found	

Mapping Operations for Biblio

- Add new items to a bibliography
- Modify or delete an item in a bibliography

Resource	Verb	Req. body	Status		Resp. body
biblio/items	POST	item	201	Created	item
			400	Bad Request	string
biblio/items/{id}	PUT	item	204	No Content	
			400	Bad Request	string
			404	Not Found	
biblio/items/{id}	DELETE		204	No Content	
			404	Not Found	

Mapping Operations for Biblio

- Modify an item in a bibliography (add citations)

Resource	Verb	Req. body	Status		Resp. body
biblio/items/{id}/citing	POST	item	204	No content	
			400	Bad Request	string
			404	Not Found	
biblio/items/{id}/cited	POST	item	204	No content	
			400	Bad Request	string
			404	Not Found	

- Missing operations (e.g., add journals and issues left as exercise)

Avoiding Large Messages

- Common best practices for responses that may be too large:
  - Query parameters to let users decide what to get
  - Paging
  - Using references instead of full representations

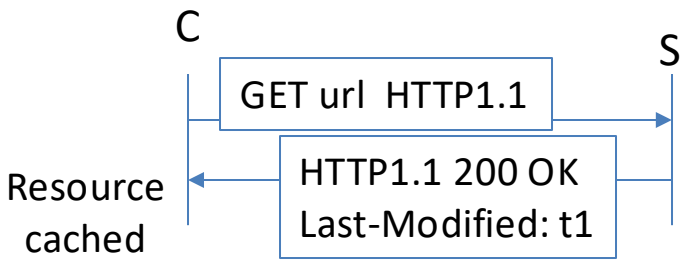
Enabling Conditional Requests

- HTTP conditional requests let clients specify that an operation should be done only under certain conditions

Header	Condition for executing
If-Modified-Since	The Last Modified date of the resource is more recent than the value of this header
If-Unmodified-Since	The Last Modified date of the resource is older than or same as the value of this header
If-Match	The Etag of the resource is equal to one of the values listed in this header
If-None-Match	The Etag of the resource is different from each of the values listed in this header

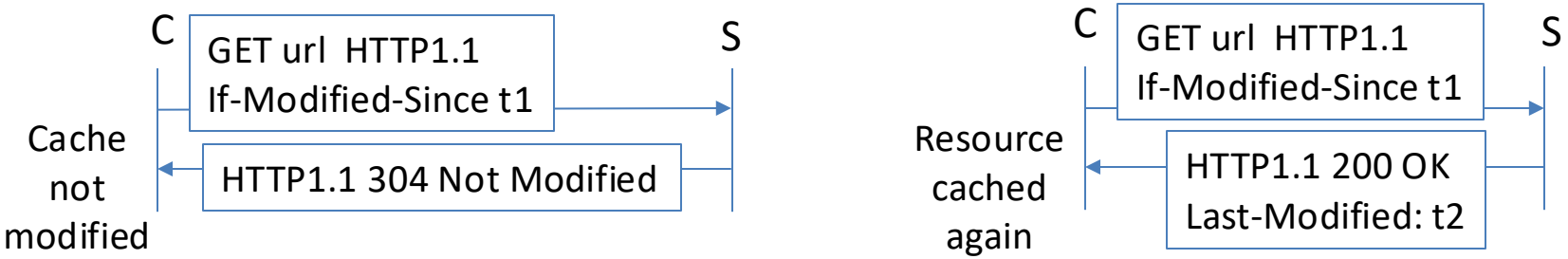
Conditional Requests Use-cases

- Management of client cache (or proxy cache)



Cache gets **stale**

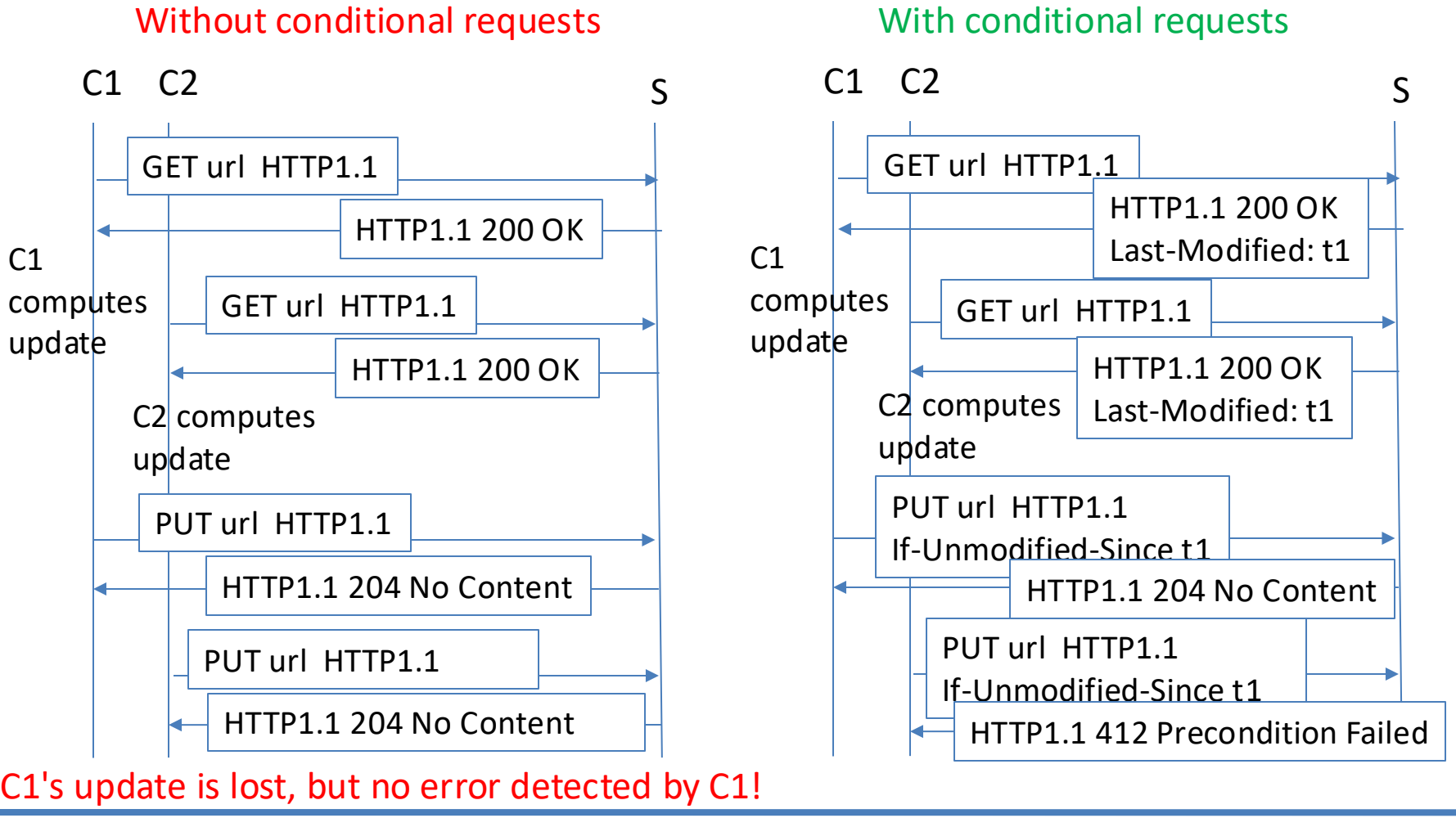
C gets resource again: Two possible cases



Cache **fresh** again

Conditional Requests Use-cases

- Avoid read-update race conditions

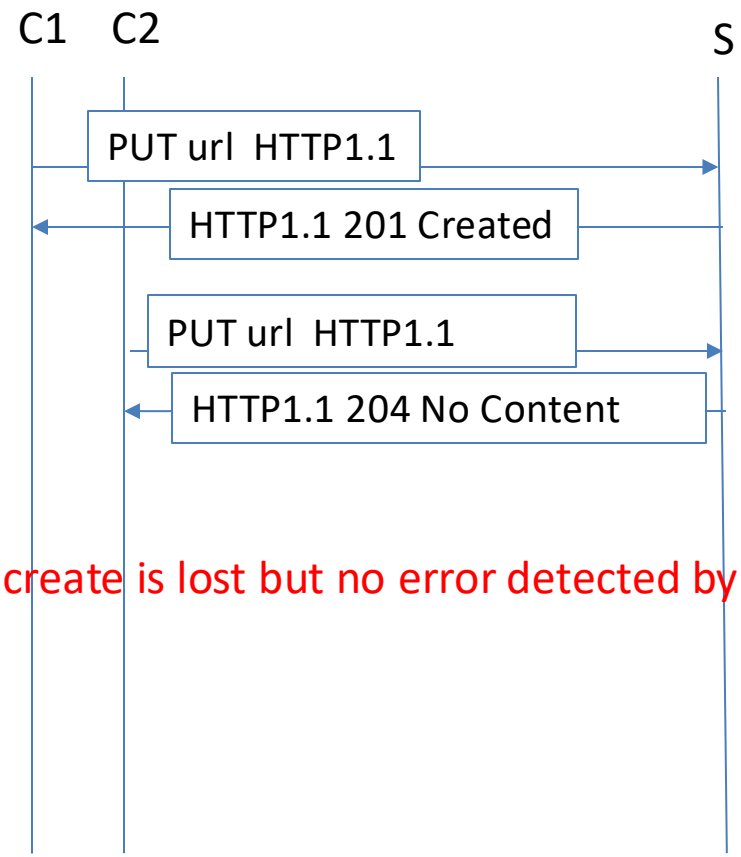


C1's update is lost, but no error detected by C1!

Conditional Requests Use-cases

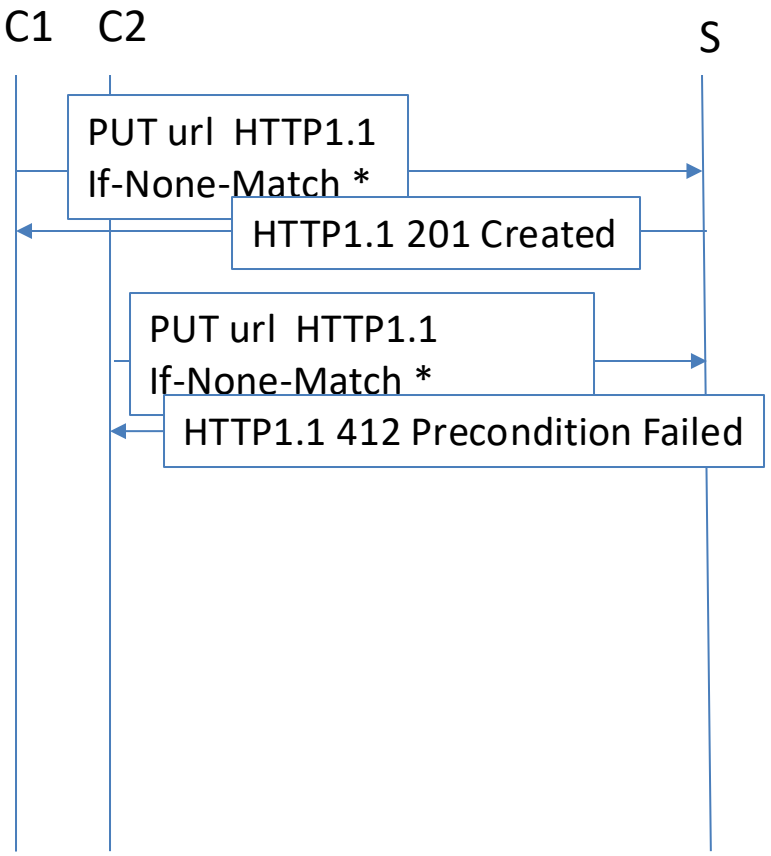
- Avoid PUT(create) race conditions

Without conditional requests



C1's create is lost but no error detected by C1!

With conditional requests



Getting HATEOAS

- Add hyperlinks to the data models of resources
  - possible alternative: add to response via HTTP link headers

```
Link: <http://example.com/users/12>; rel=next
```

- Assume clients should never "build" hyperlinks, but just follow the ones provided in the responses
- Stick to conventions that make the clients automatically understand the meaning of hyperlinks
  - "self" hyperlink points to the resource itself
  - "next" hyperlink is used when navigating to the next resource in a sequence
  - ... https://www.iana.org/assignments/link-relations/link-relations.xhtml