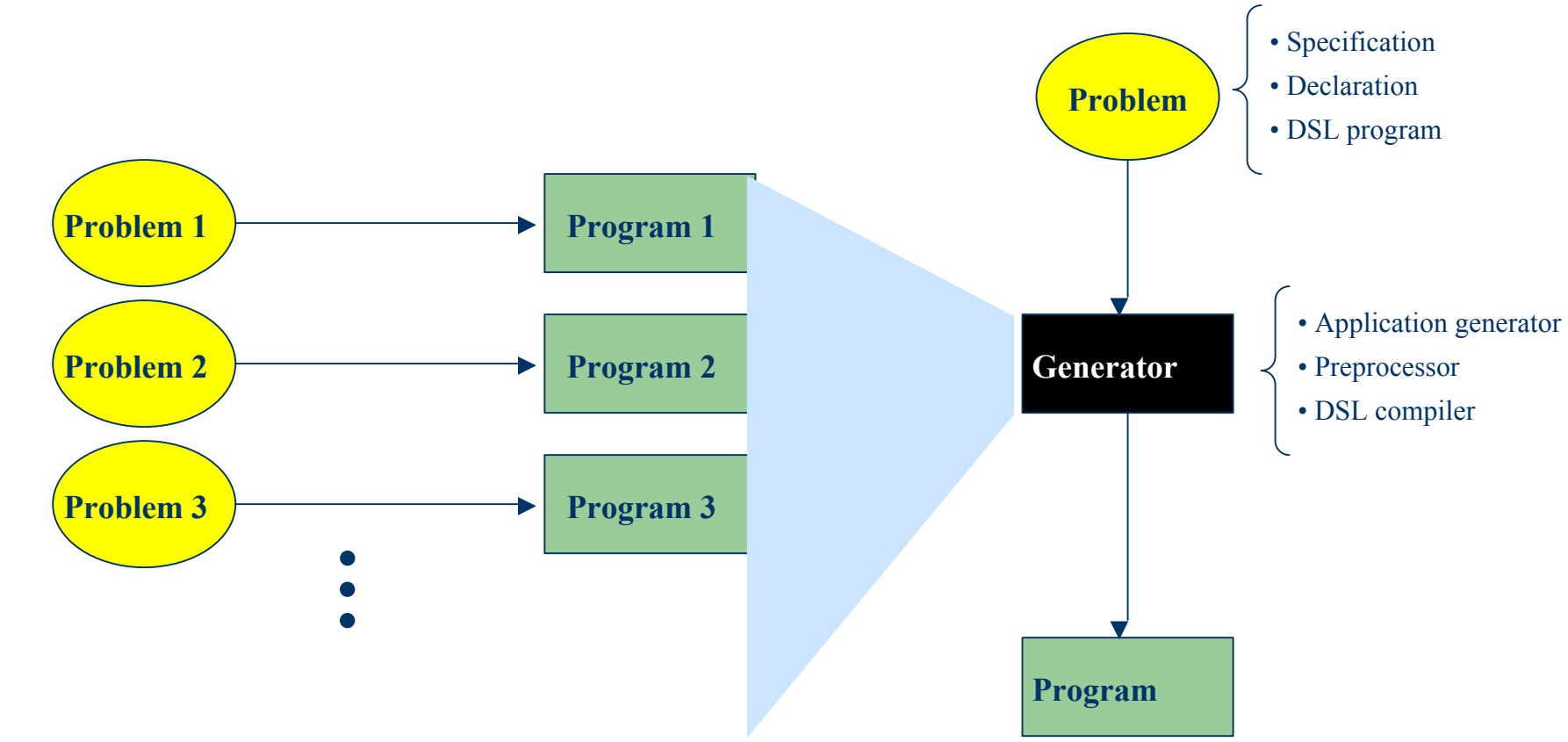
Toward Structuring The Development Of A DSL Using A Program Family

Charles Consel
(joint work with many members of Compose)
Compose Research Group
INRIA -LaBRI

September 2003

Domain-Specific Languages: The Basic Idea

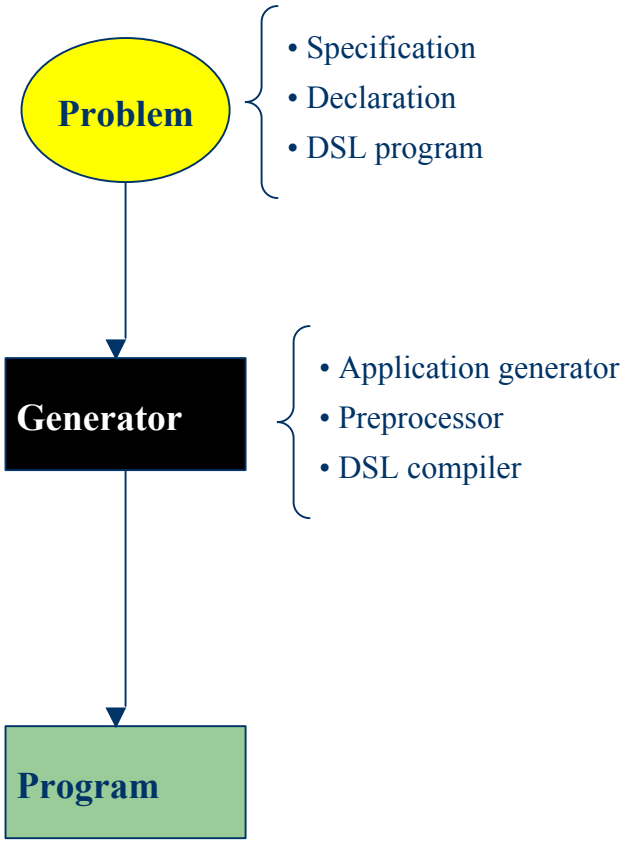


Domain-Specific Languages: Issues

- When to develop a DSL?
- What is the scope of the DSL?
- How to design the DSL?
  - Key concept: program family

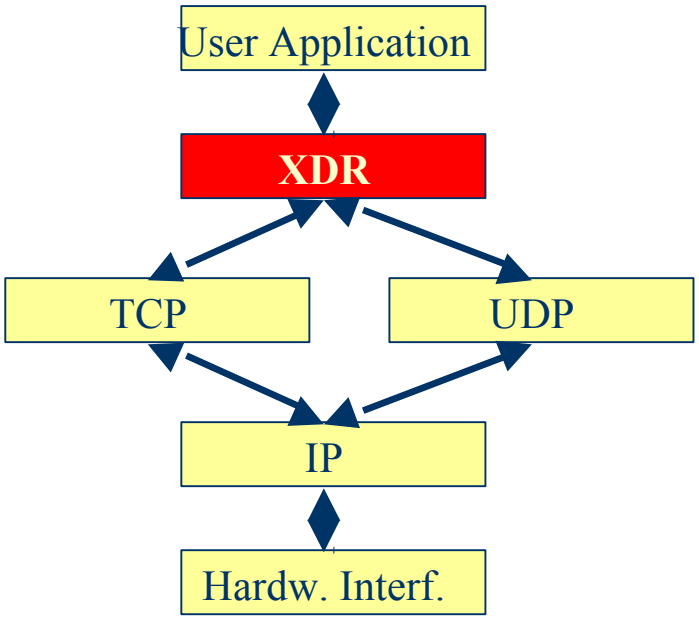
- How to implement the DSL?
  - Approach: stepwise methodology
    - Data coding/marshalling
    - Device interface layer

- How to assess the benefits?
  - Key concept: program family



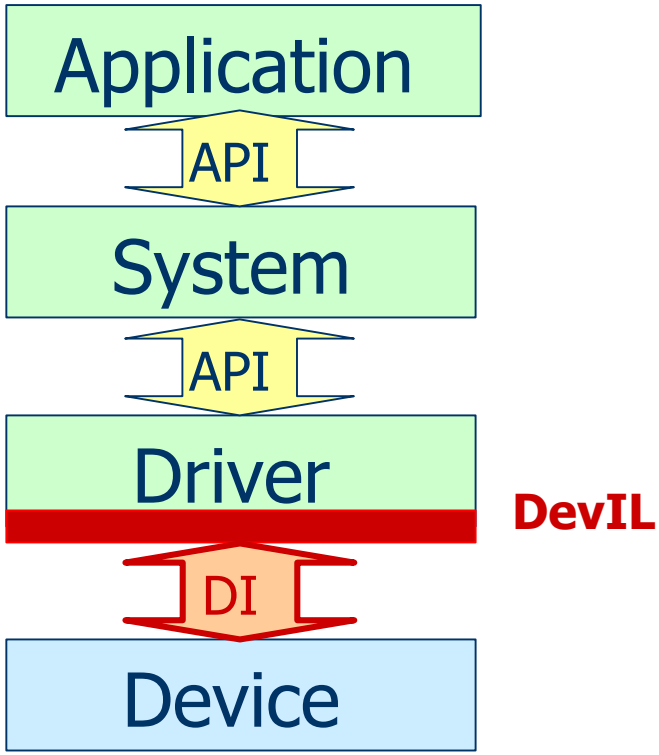
A Program Family with a DSL: Machine Independent Data Coding

- Commonalities: data traversal, type description…
- Variations: coding direction, size, format…



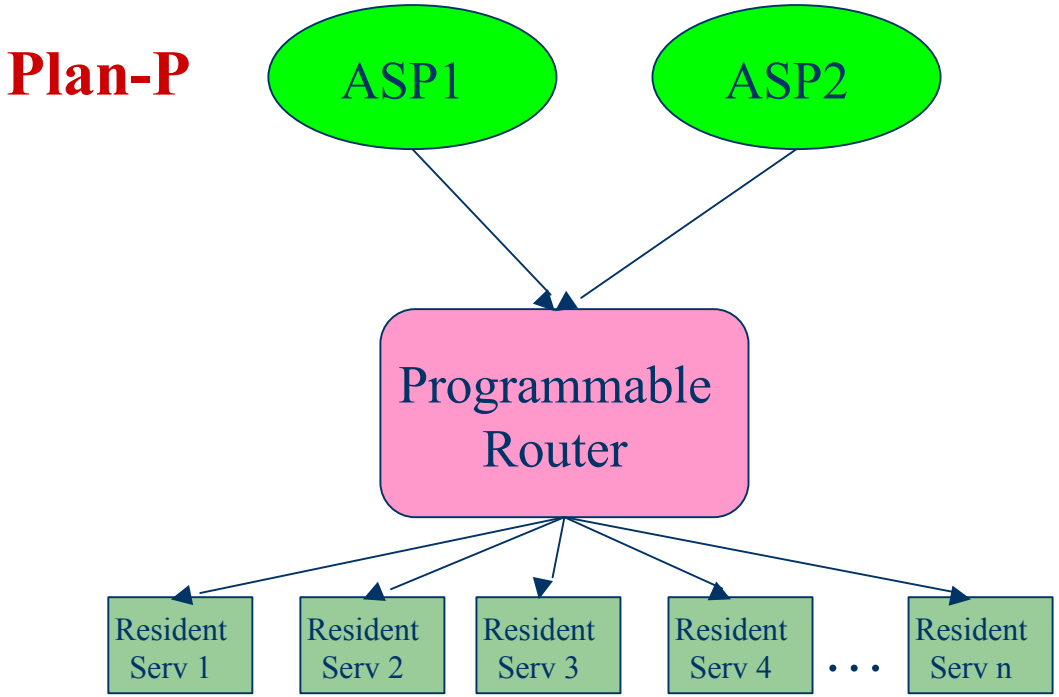
A Program Family with our DSL: Device Driver Interface

- Commonalities: API, bit operations...
- Variations: parameters, registers…



A Program Family with our DSL: Application-Specific Protocols

- Commonalities: API, packets operations
- Variations: routing policies, packet processing…



Program Family Characterization: A Key Step Towards DSL Design

- Nature of program family
  - Existing(Devil)
  - Virtual (~ Plan-P)
- Scope of programs
  - Layers (Devil)
  - Components (Plan-P)
- Range of commonalities
  (size of program family captured)
- Scope of computations
  (kind of expressible computations)

Beyond Program Families

- Technical literature
- Documentation
- Domain experts
- Current and future requirements

Domain-Specific Languages: A (Conceptual) Definition

- Program family
- Domain-specific abstractions and notations

  Conciseness, readability
- Declarative (often)

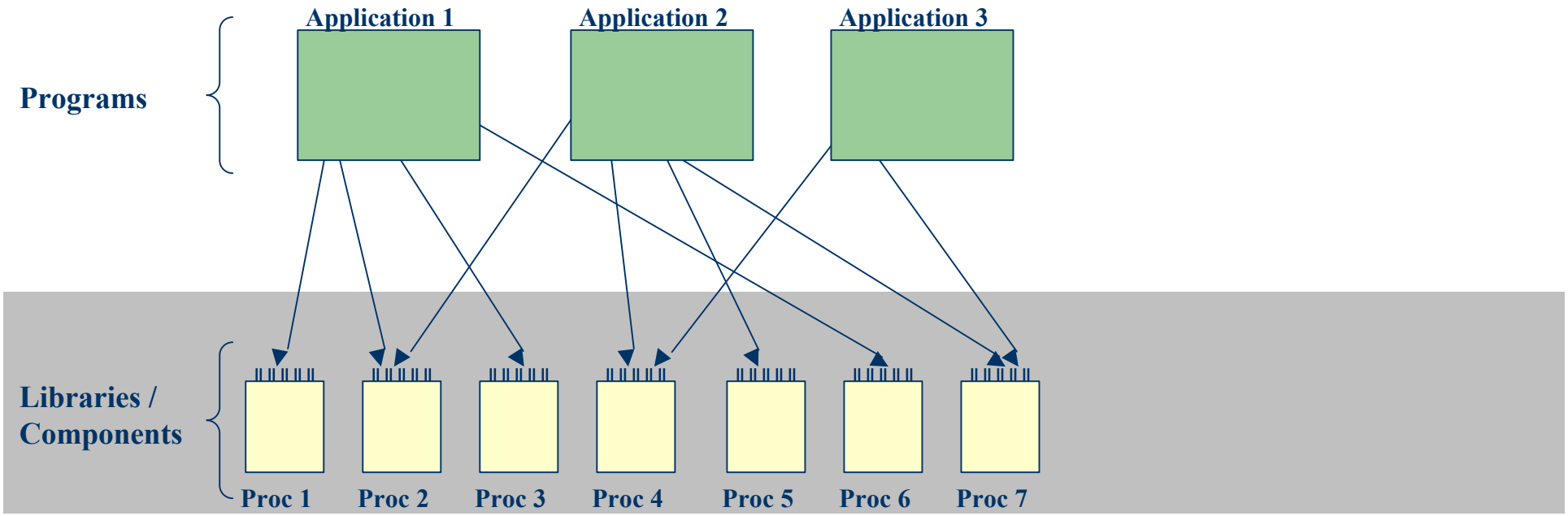
  What to compute, not how to compute it
- Restricted/enriched semantics
  - Program verification: making critical properties decidable
  - Program optimization: customizing implementation

Our Approach to DSLs

- DSL structure
  - Static semantics: interpreter/compiler
  - Dynamic semantics: Abstract machine
- Stepwise methodology:
  - Program family
  - Library/components
  - Abstract machine
  - DSL

Program Family: Libraries / Components

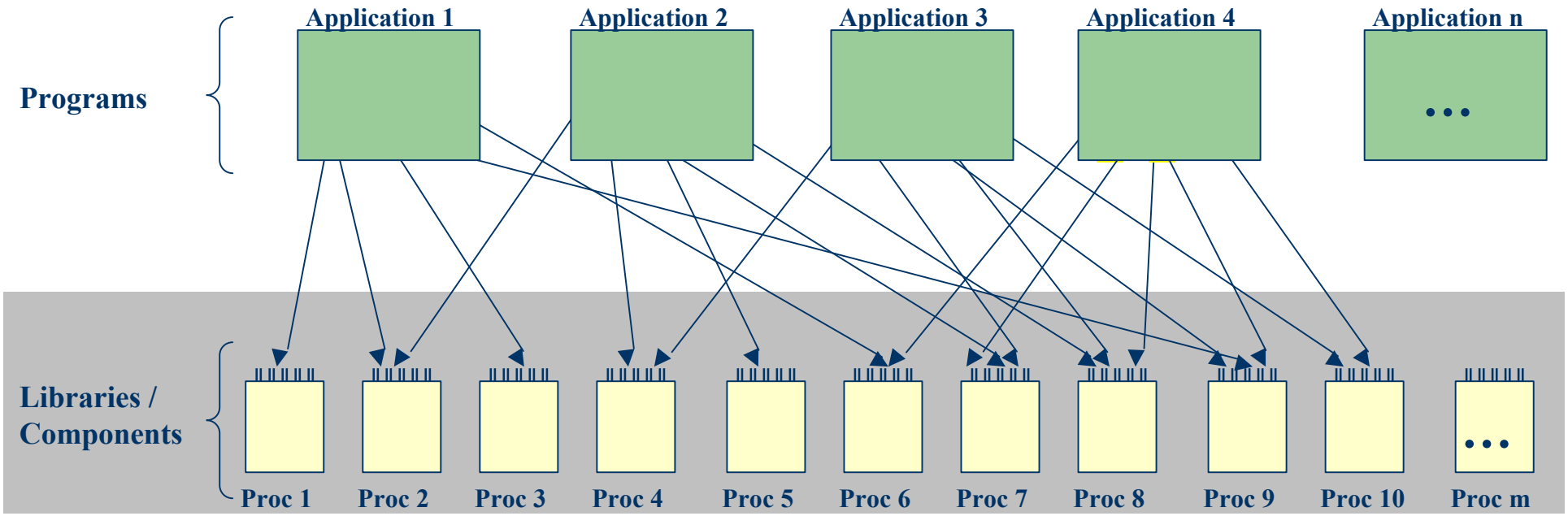
- Commonalities ⇨ Library entries
- Variations ⇨ Parameters



Program Family: Libraries / Components

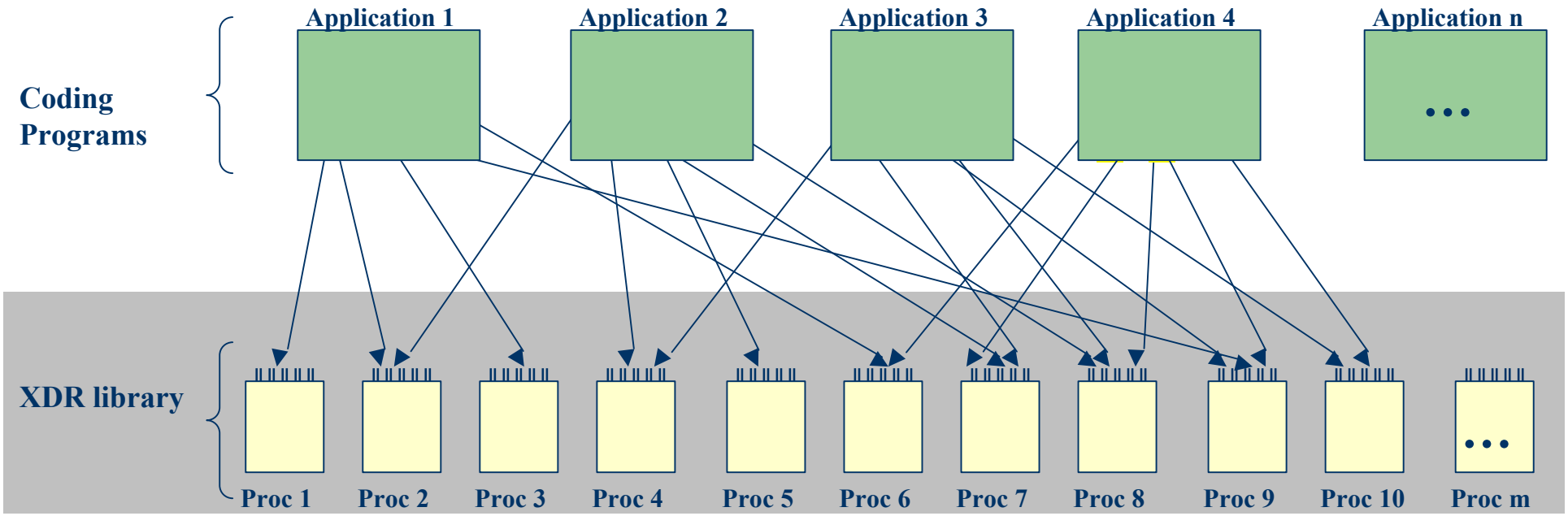
- More commonalities ⇨ More entries
- More variations ⇨ More parameters

- Poor performance
  - Interpretation layer
- Bloated code
  - Numerous cases



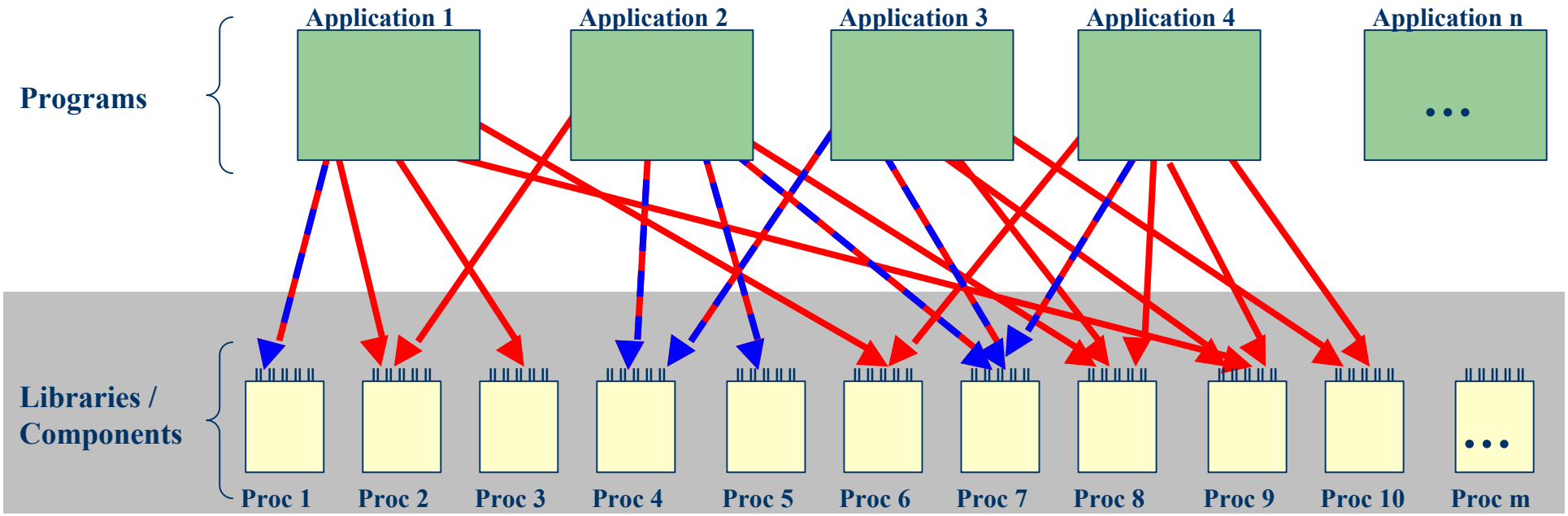
Coding Programs: XDR Library

`xdr_int`	Integer size
`xdr_long`	Coding direction
`XDR_PUTLONG`	Generic marshalling
`xdrmem_putlong`	Buffer write + overflow check
`htonl`	Byte ordering



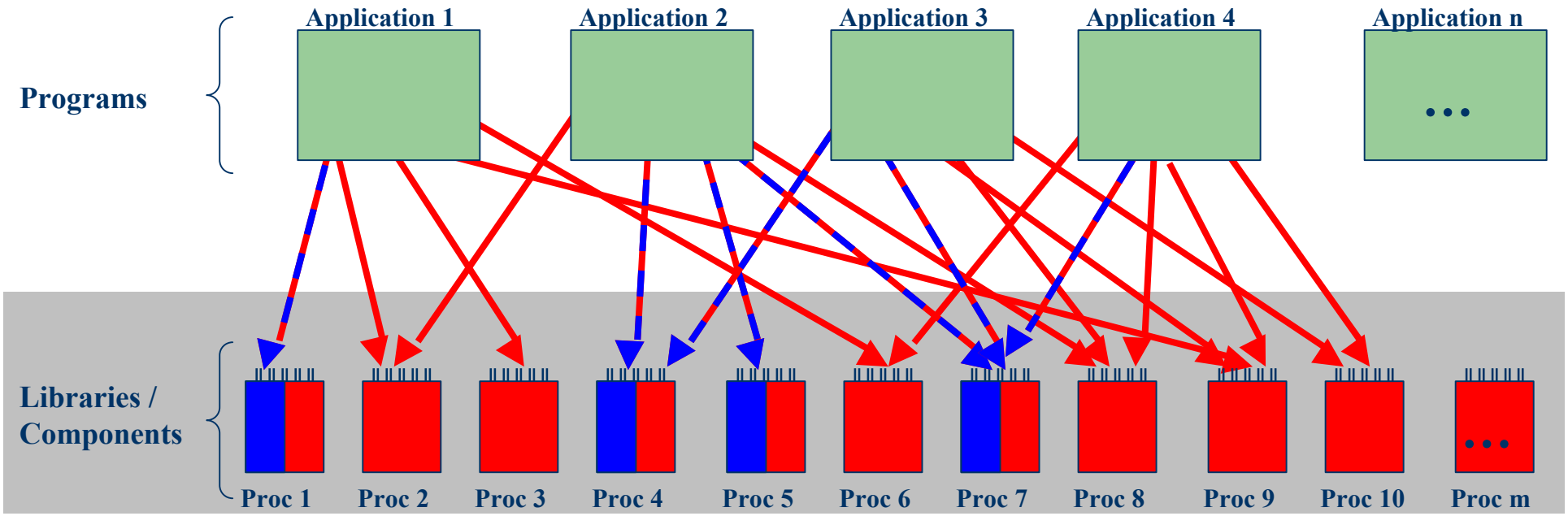
Program Family: Libraries / Components

- For a given set of problems : ***invariants***



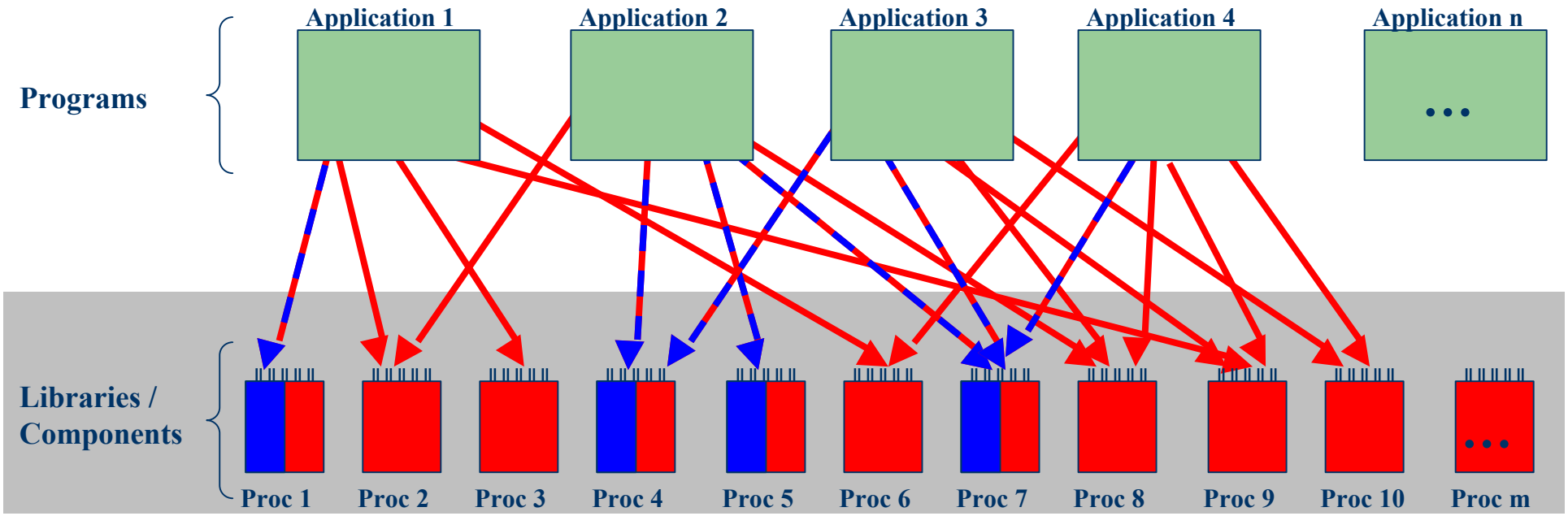
Libraries/Components :
Need For Customization

- For a given set of problems : ***invariants***
  - Optimize time (and/or)
  - Optimize space



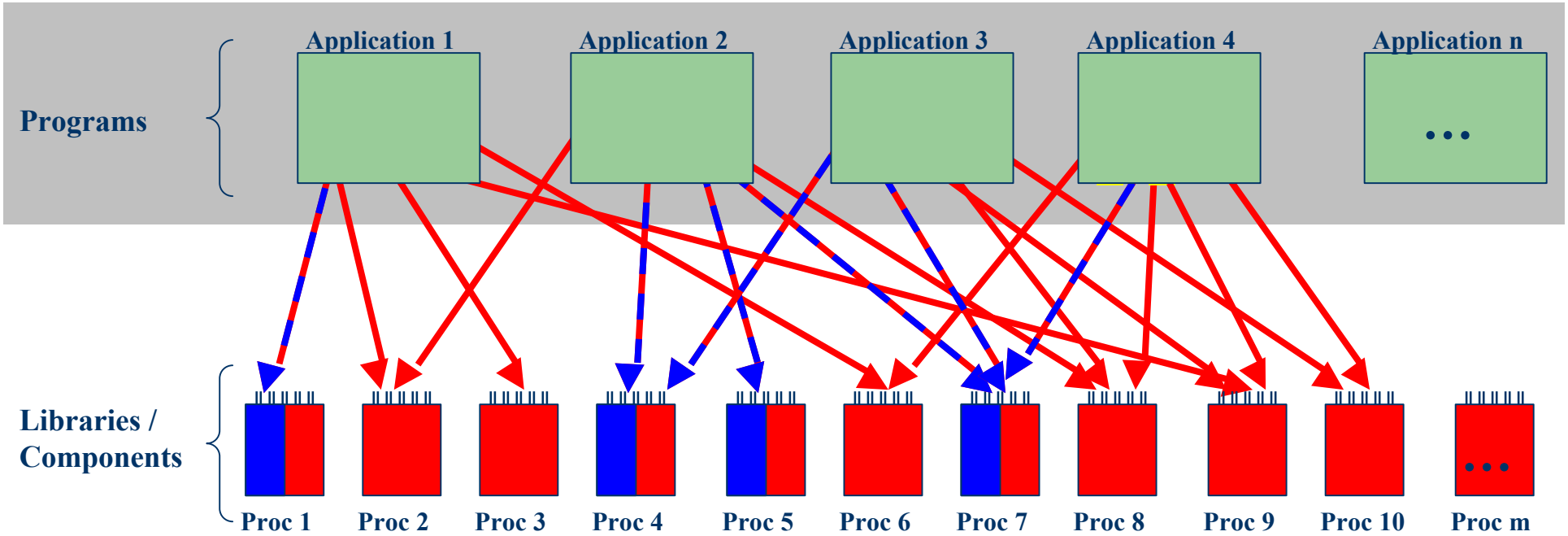
XDR Library :
Need For Customization

`xdr_int`	Integer size
`xdr_long`	Coding direction
`XDR_PUTLONG`	Generic marshalling
`xdrmem_putlong`	Buffer write + overflow check
`htonl`	Byte ordering



Program Family: Applications

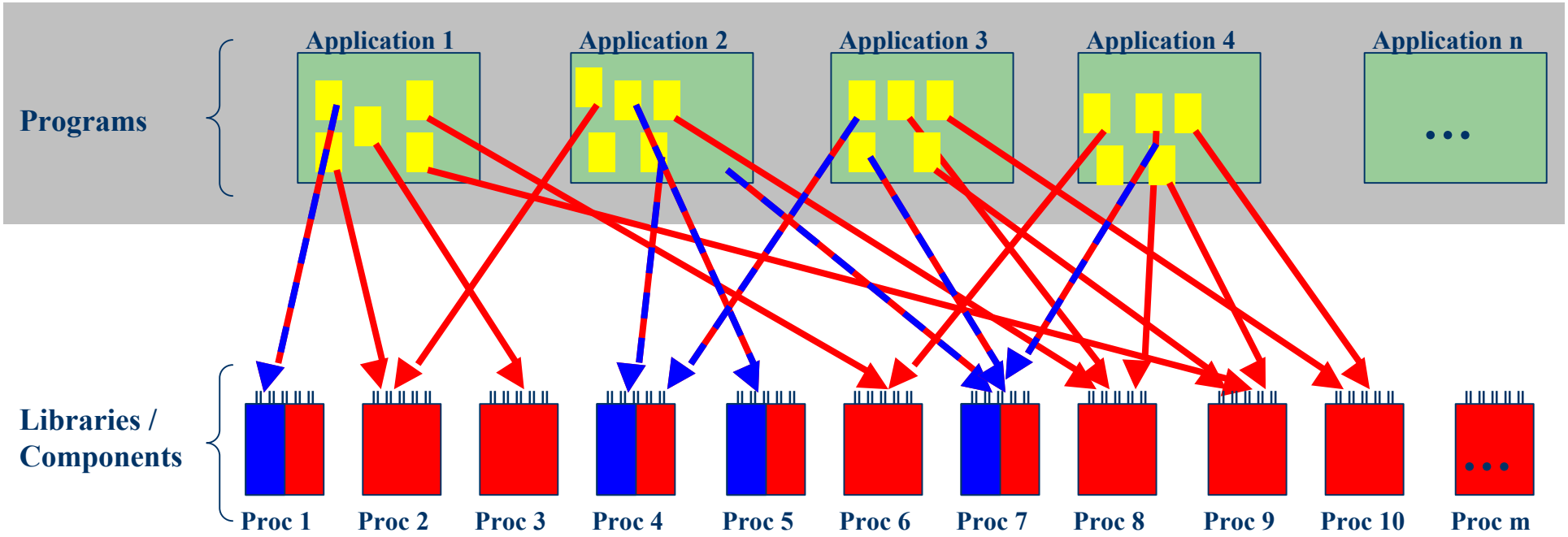
- General-purpose language
- Lack of software architecture



Program Family: Applications

- General-purpose language
- Lack of software architecture

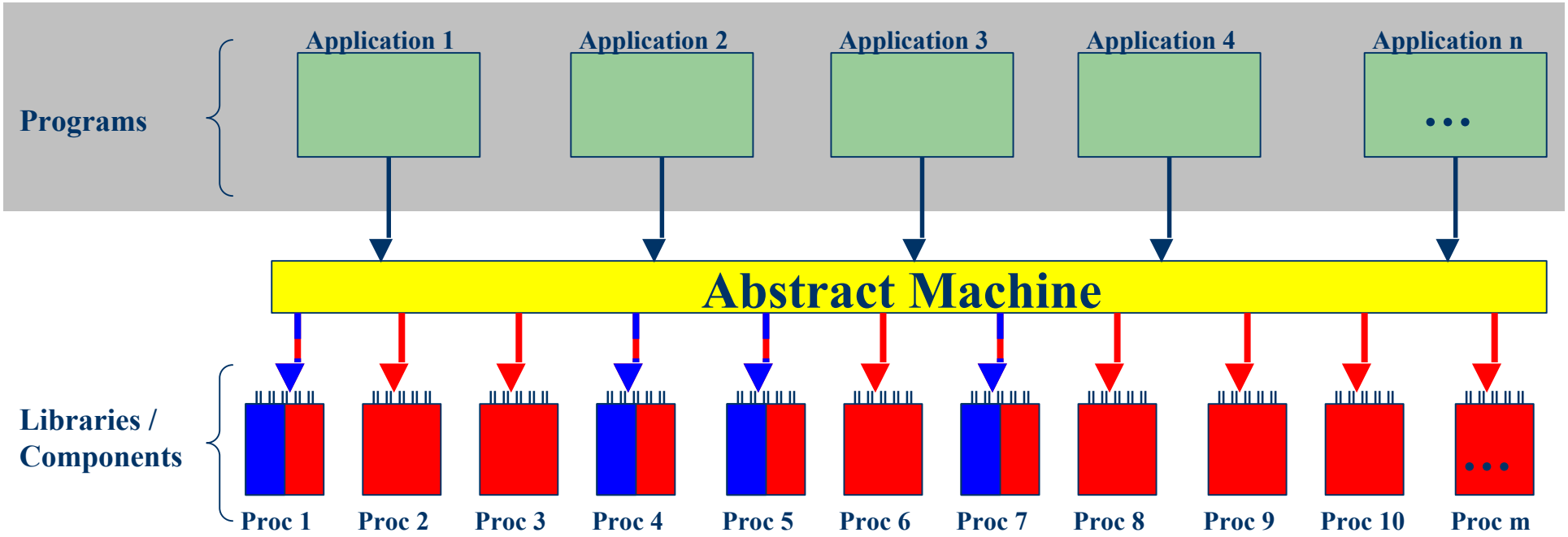
- Prologues/epilogues
  - Prepare invocations
  - Maintain state
  - Perform checks
- Results: complex, repetitive, expertise required



Applications:
Need for Abstract Machine *(intermediate step)*

- Interface to libraries
- Explicit run-time model
  - State definition and operations
  - Dynamic checks
  - Operation encapsulation
  - Operation combination

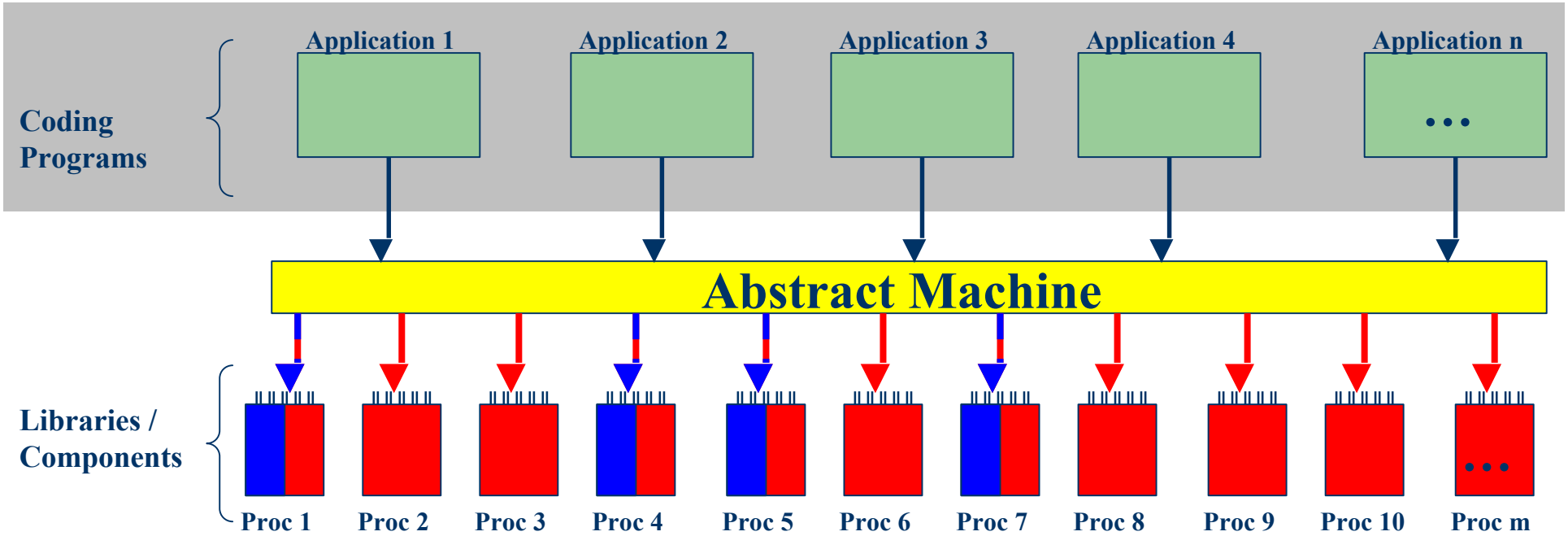
- More concise
- More usable
- Factorized expertise
- Abstraction layer



Coding Applications:
Need for Abstract Machine *(intermediate step)*

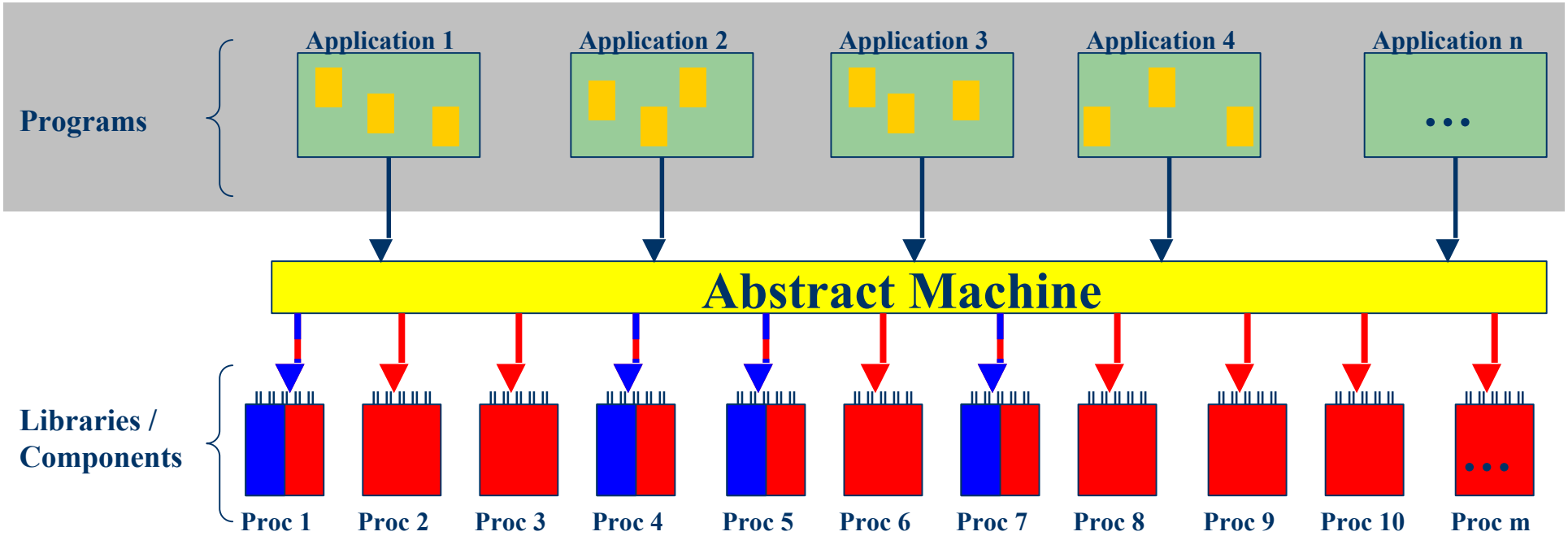
- More concise
- More usable
- More explicit expertise
- Abstraction layer

- **XDR** Data Coding Machine
  - Instruction set: XDR library
  - State: XDR structure
  - Parameterized instruction set



Program Family: Applications

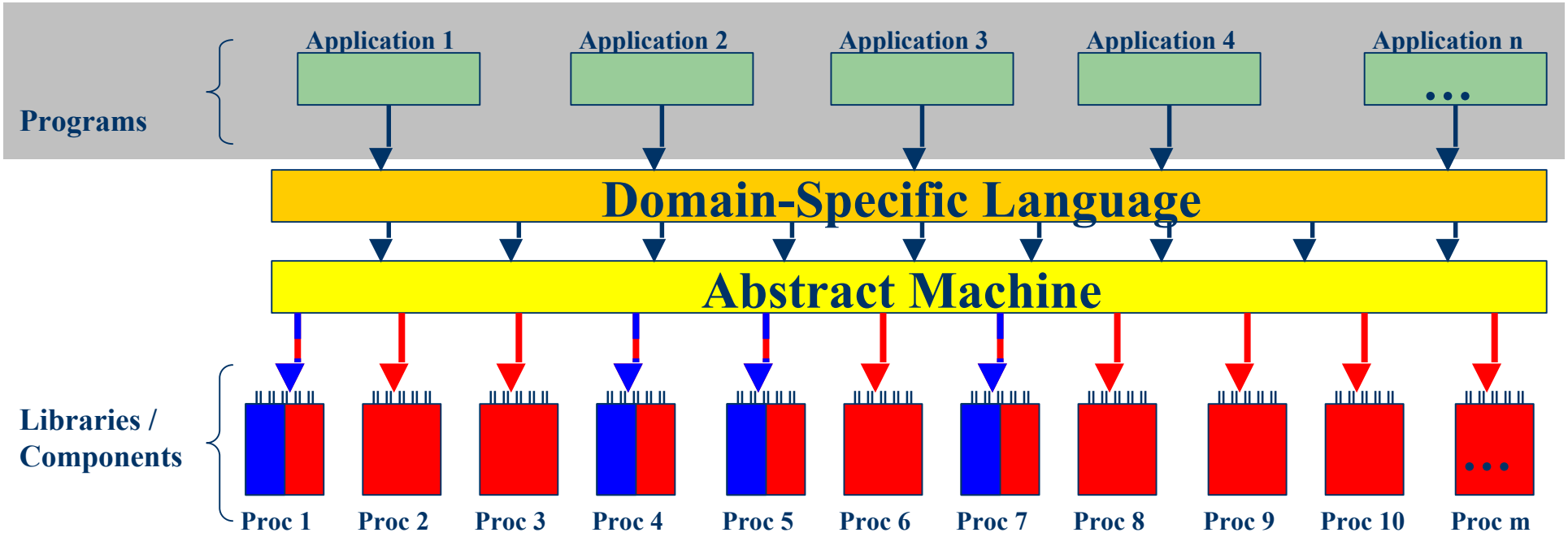
- General-purpose language
  - For each call
    - Prologues/epilogues (still)
  - For each call sequence
    - Repetitive/*required* call patterns
    - Repetitive/*required* checks



Applications:
Need for Domain-Specific Language

- Interface abstract machine
- Designate a family member
- Domain specific constructs (abstractions/notations)
- Restricted/enriched semantics

- Re-use
- Expertise
- Conciseness
- Verification
- Performance
- Productivity

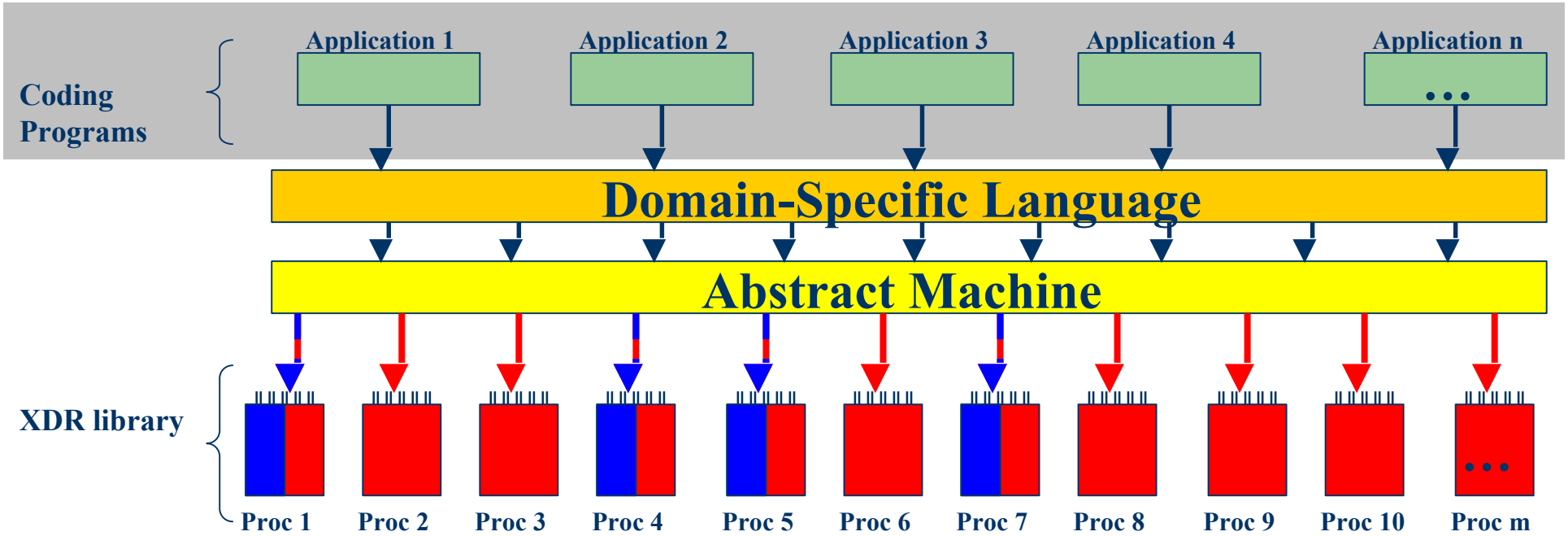


Key concept:
Data Structure Descriptions

XDR language

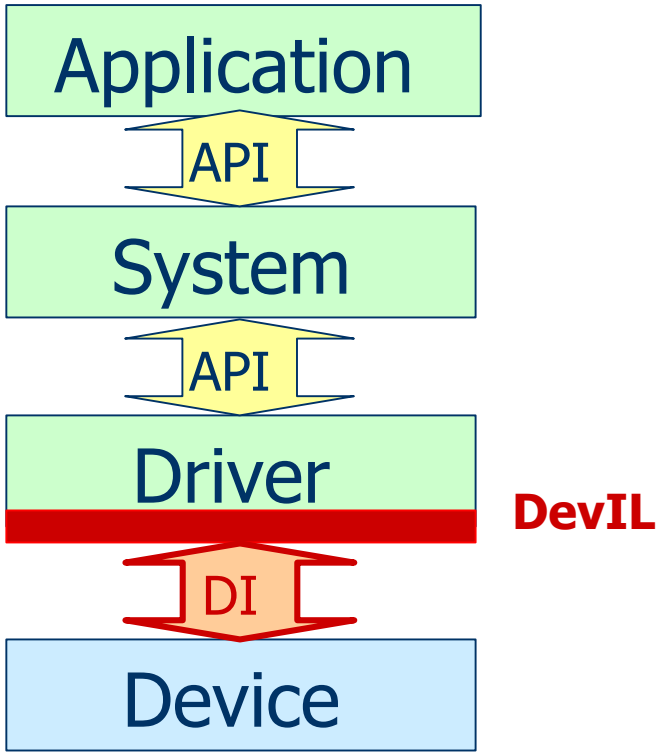
```
union result switch (int error){
  case 0:
     char data[MAX_SIZE];
  default:
     void;
}
```

- Re-use
- Expertise
- Conciseness
- Verification
- Productivity
- Performance

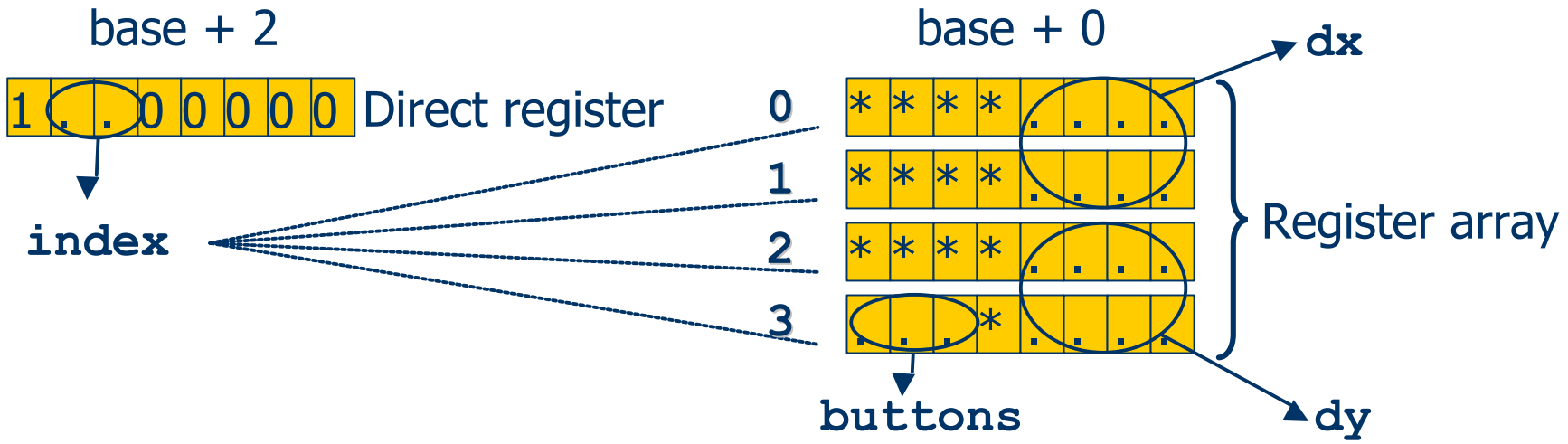


A Program Family with our DSL: Device Driver Interface

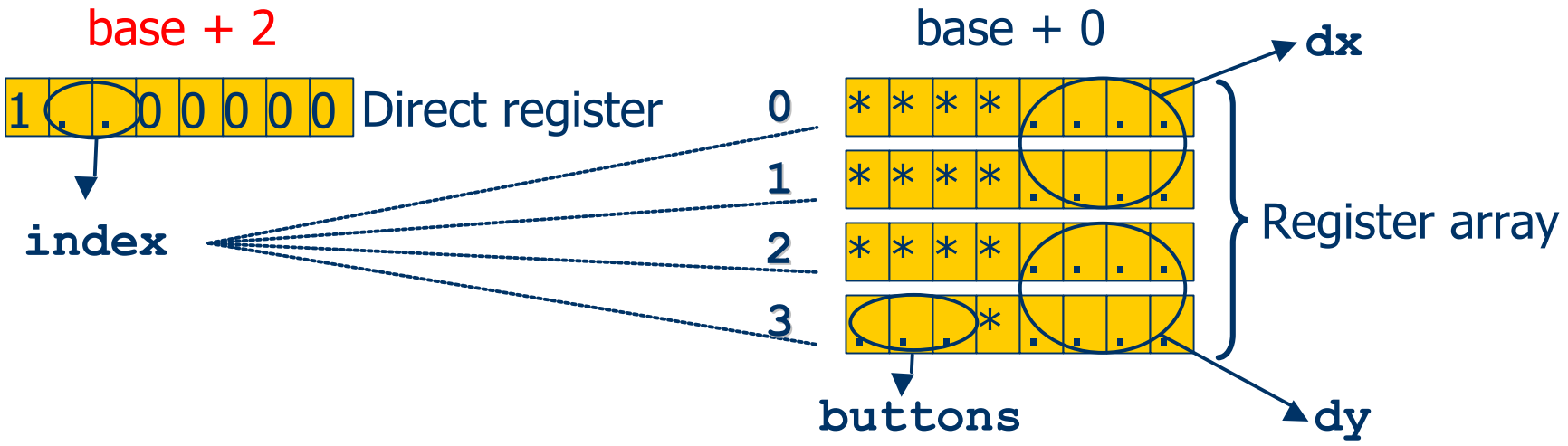
- Commonalities: API, bit operations...
- Variations: parameters, registers…



Driver Example : Linux mouse



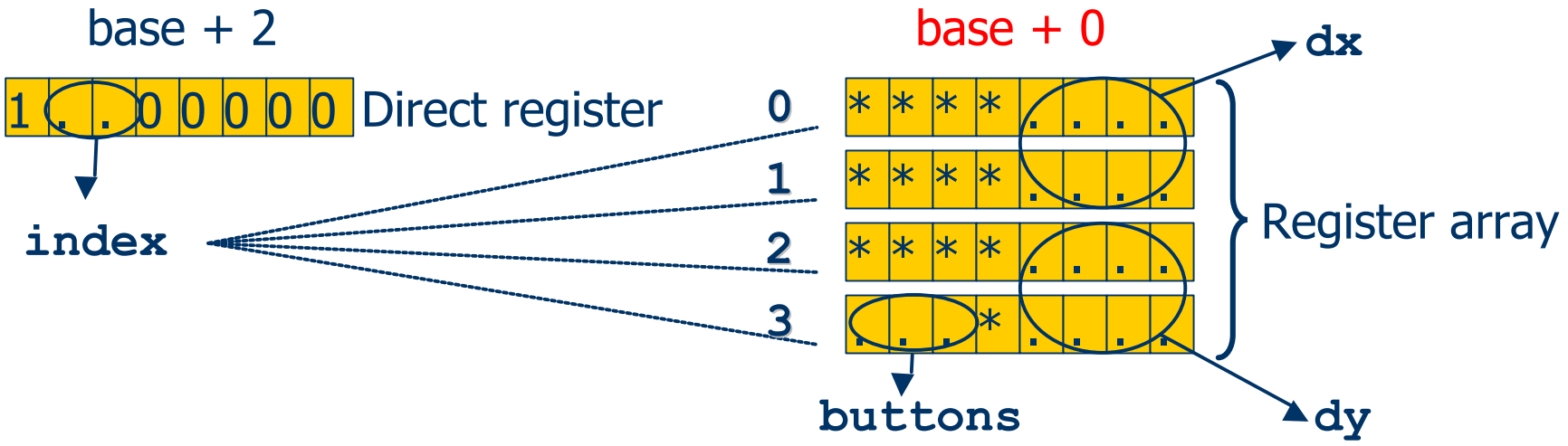
Driver Example : Linux mouse



(busmouse.h)

```
#define MSE_CONTROL_PORT       0x23e
```

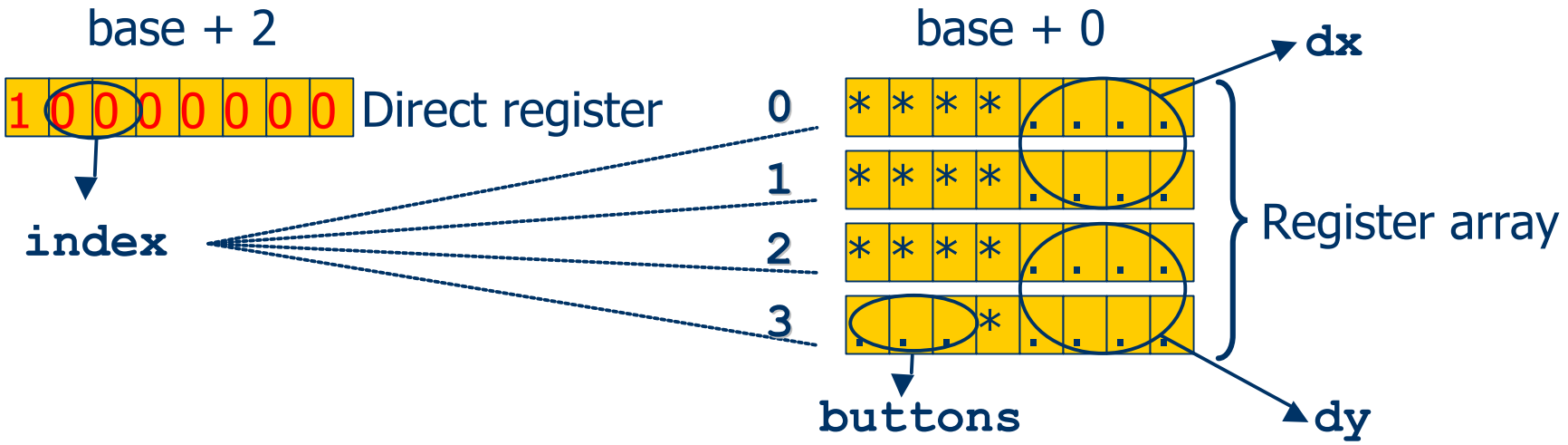
Driver Example : Linux mouse



(busmouse.h)

```
#define MSE_CONTROL_PORT        0x23e
#define MSE_DATA_PORT           0x23c
```

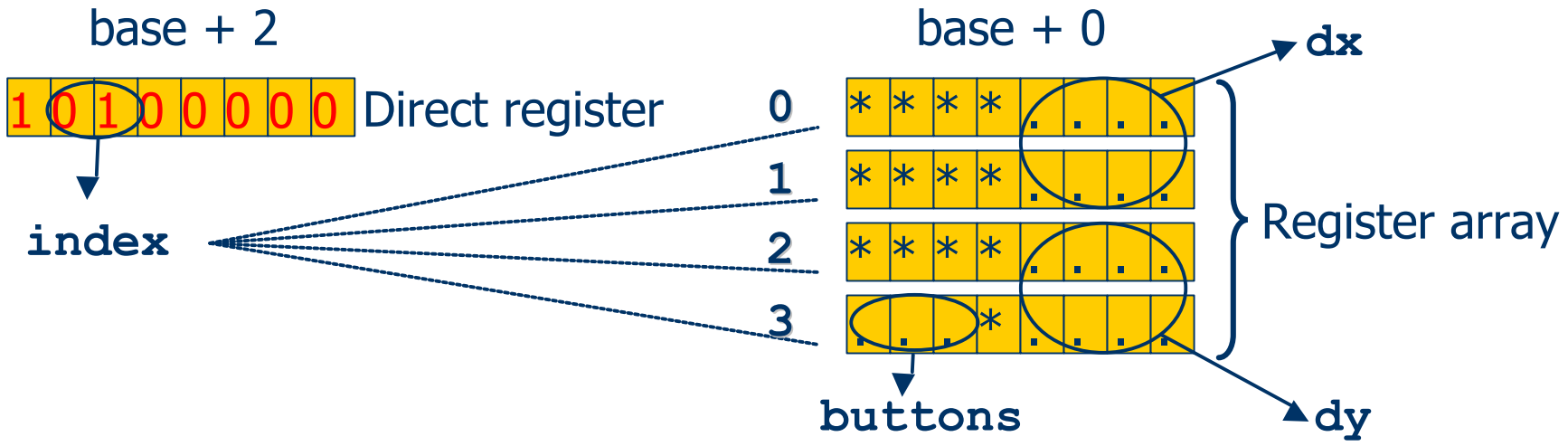
Driver Example : Linux mouse



(busmouse.h)

```
#define MSE_DATA_PORT          0x23e
#define MSE_CONTROL_PORT       0x23c
#define MSE_READ_X_LOW         0x80
```

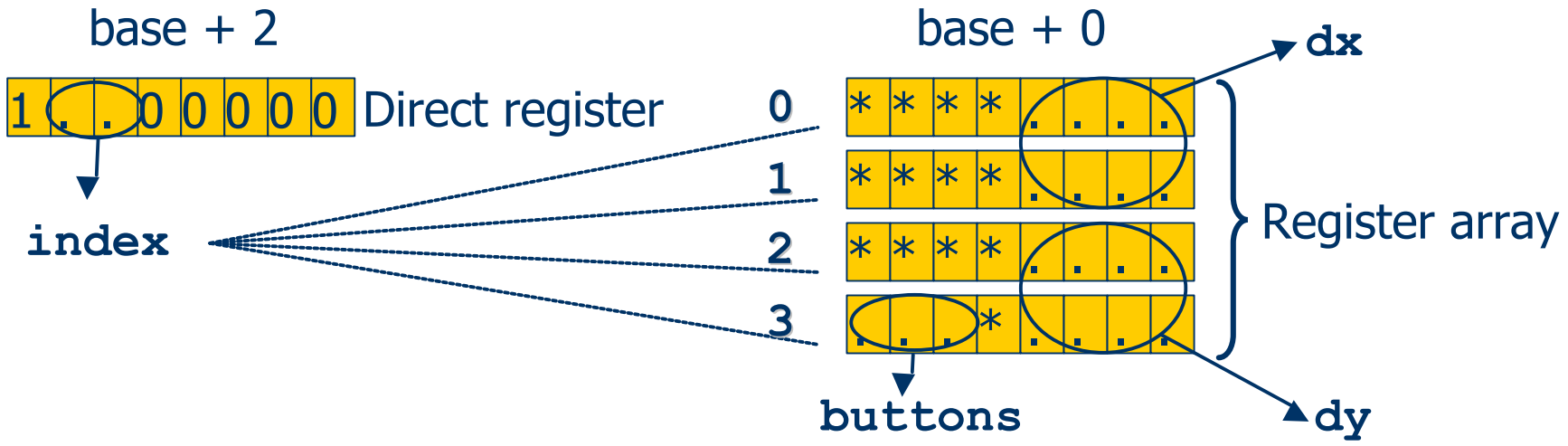
Driver Example : Linux mouse



(busmouse.h)

```
#define MSE_DATA_PORT          0x23e
#define MSE_CONTROL_PORT       0x23c
#define MSE_READ_X_LOW         0x80
#define MSE_READ_X_HIGH        0xa0
```

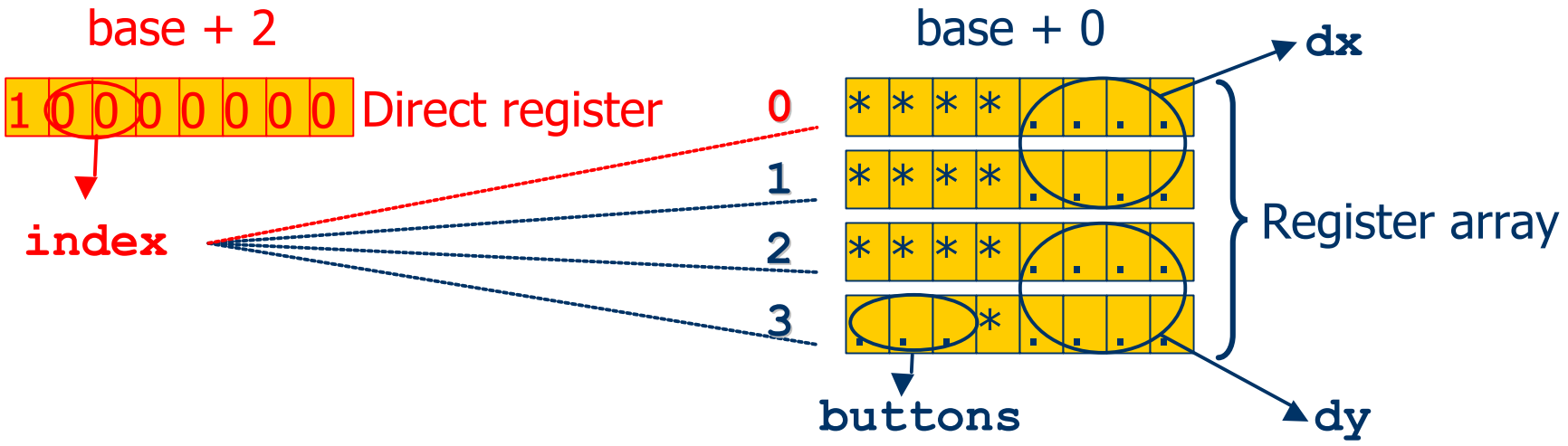
Driver Example : Linux mouse



(busmouse.h)

```
#define MSE_DATA_PORT          0x23e
#define MSE_CONTROL_PORT       0x23c
#define MSE_READ_X_LOW         0x80
#define MSE_READ_X_HIGH        0xa0
...
```

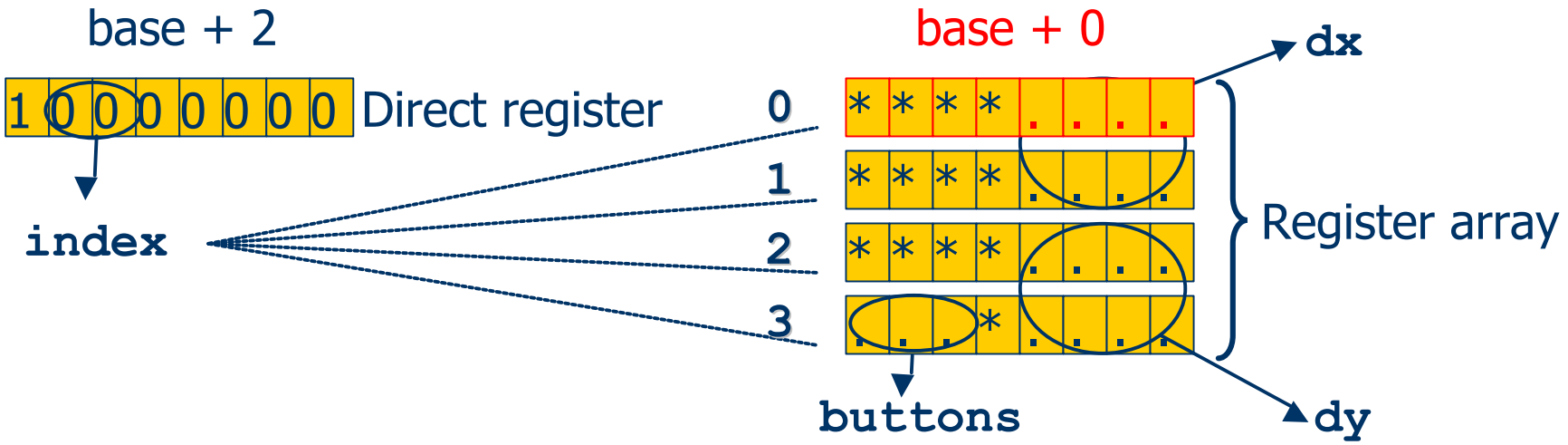
Driver Example : Linux mouse



(busmouse.c)

```
outb(MSE_READ_X_LOW, MSE_CONTROL_PORT);
```

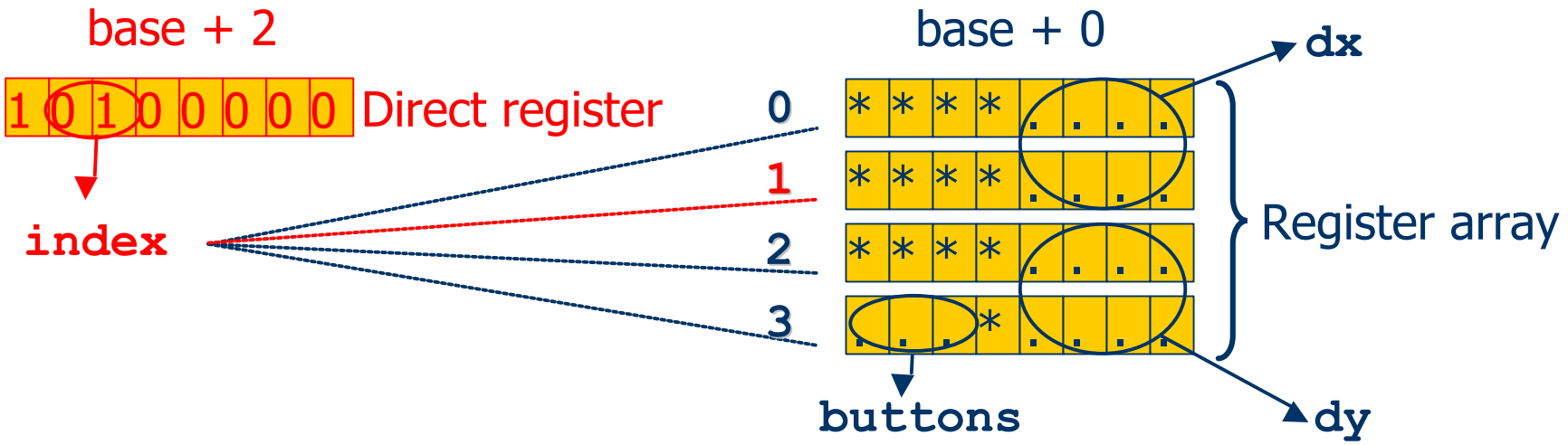
Driver Example : Linux mouse



(busmouse.c)

```
outb(MSE_READ_X_LOW, MSE_CONTROL_PORT);
dx = (inb(MSE_DATA_PORT) & 0xf);
```

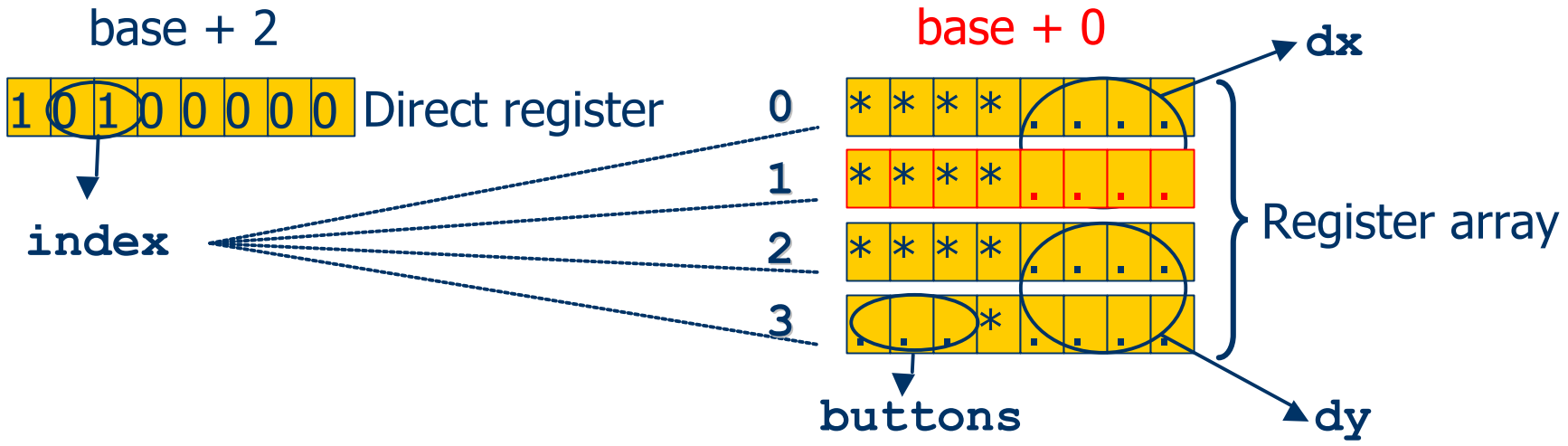
Driver Example : Linux mouse



(busmouse.c)

```
outb(MSE_READ_X_LOW, MSE_CONTROL_PORT);
dx = (inb(MSE_DATA_PORT) & 0xf);
outb(MSE_READ_X_HIGH, MSE_CONTROL_PORT);
```

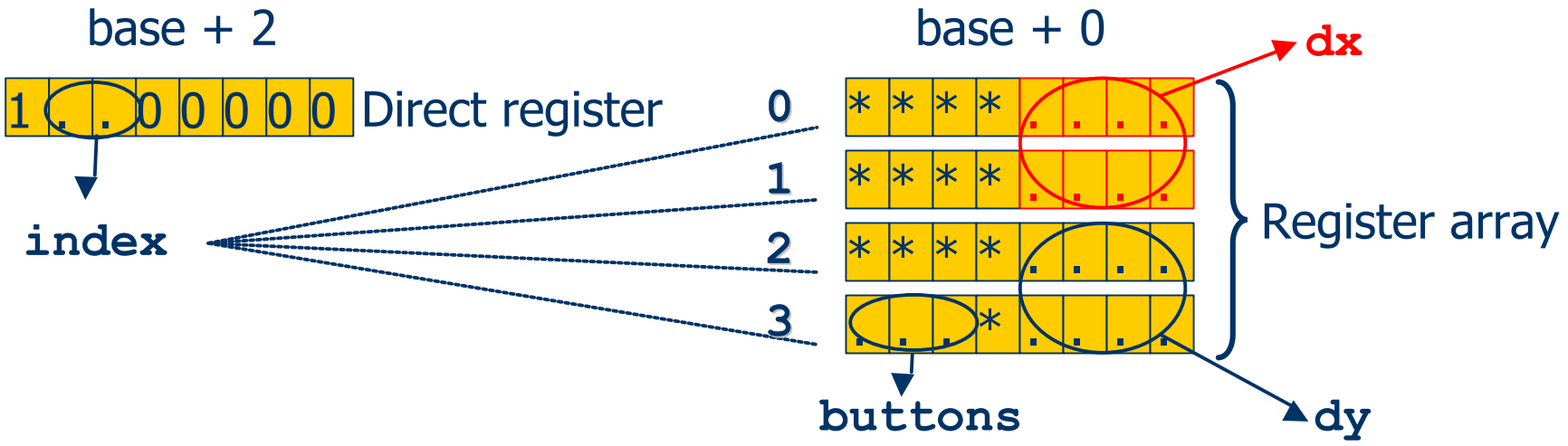
Driver Example : Linux mouse



(busmouse.c)

```
outb(MSE_READ_X_LOW, MSE_CONTROL_PORT);
dx = (inb(MSE_DATA_PORT) & 0xf);
outb(MSE_READ_X_HIGH, MSE_CONTROL_PORT);
dx |= (inb(MSE_DATA_PORT) & 0xf) << 4;
```

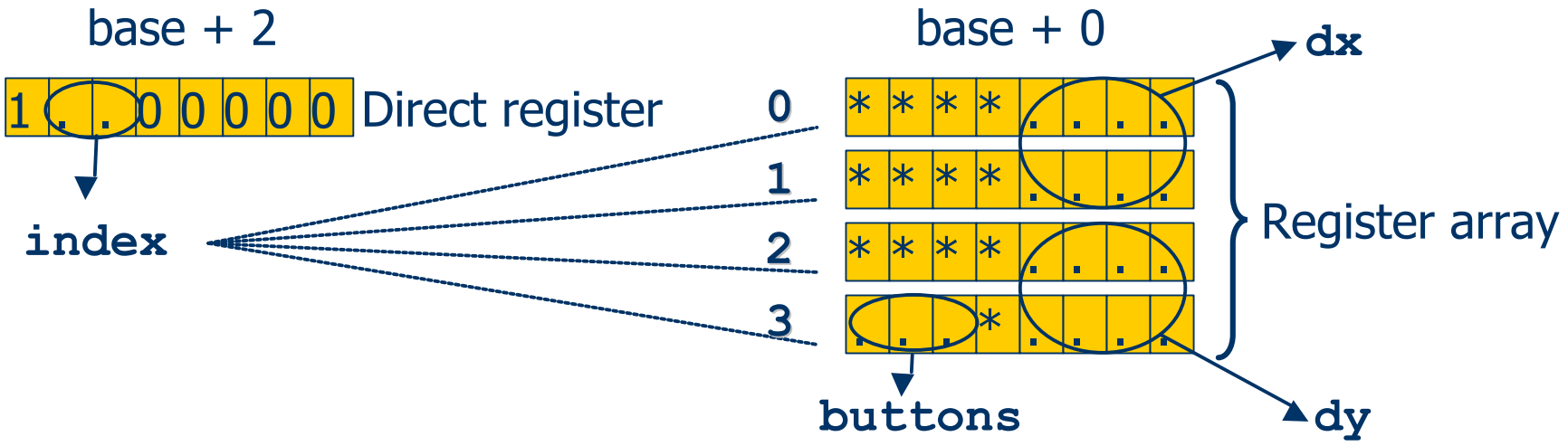
Driver Example : Linux mouse



(busmouse.c)

```
outb(MSE_READ_X_LOW, MSE_CONTROL_PORT);
dx = (inb(MSE_DATA_PORT) & 0xf);
outb(MSE_READ_X_HIGH, MSE_CONTROL_PORT);
dx |= (inb(MSE_DATA_PORT) & 0xf) << 4;
```

Driver Example : Linux mouse



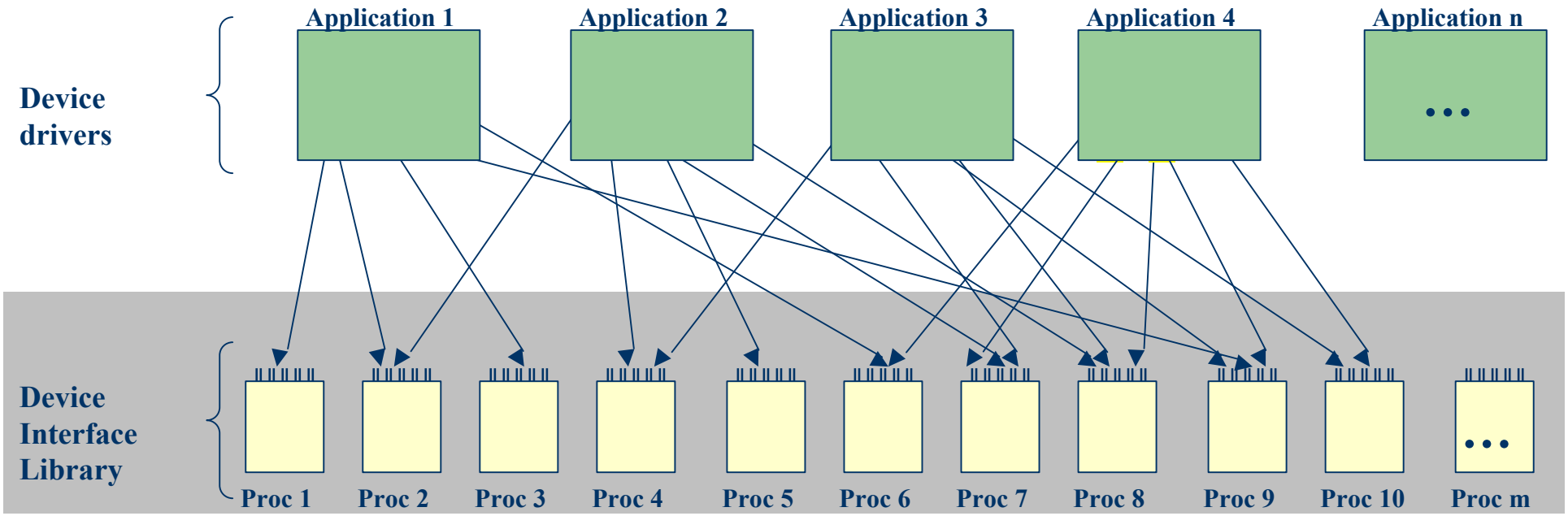
(busmouse.c)

```
outb(MSE_READ_X_LOW, MSE_CONTROL_PORT);
dx = (inb(MSE_DATA_PORT) & 0xf);
outb(MSE_READ_X_HIGH, MSE_CONTROL_PORT);
dx |= (inb(MSE_DATA_PORT) & 0xf) << 4;
...
```

Family of Layers: Device Interface Library

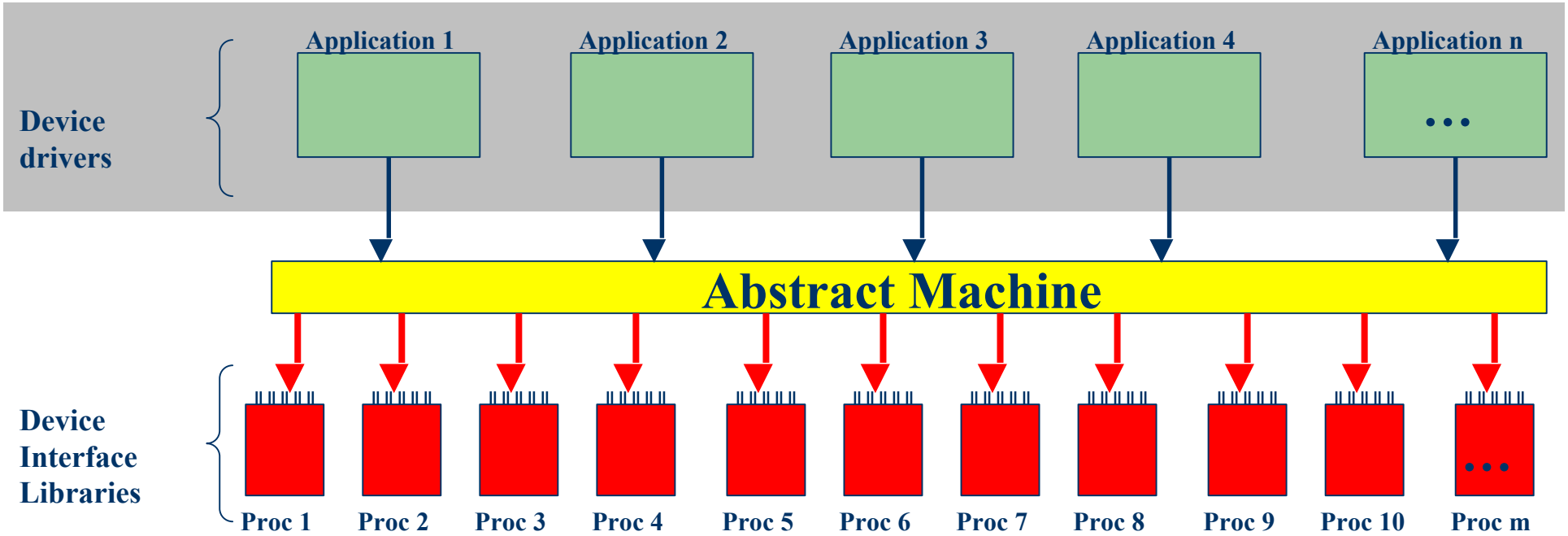
- Bit operations
- Input/output operations

- Low level
- Fine grained
- Directly mapped into machine instructions
- Implicit state



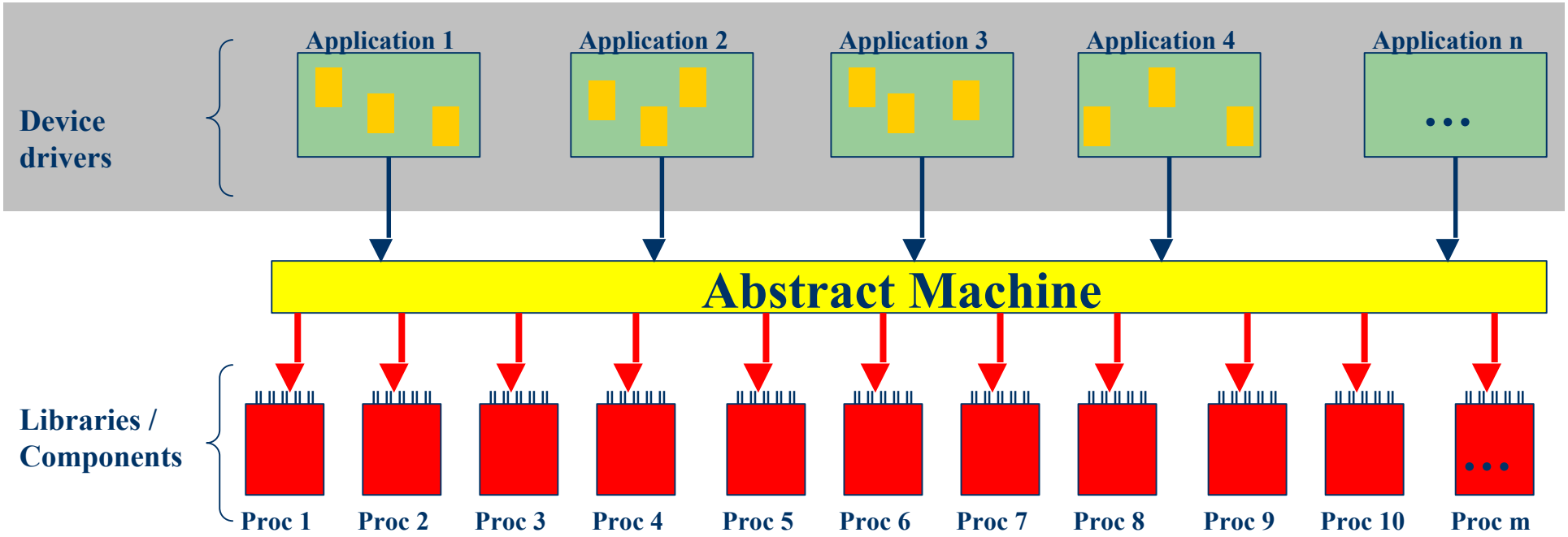
Applications:
Device Interface Abstract Machine

- Bit extraction operations
- Bit concatenation operation
- Input/output abstract operations
- Input/output checks
- Dedicated set of bit instructions
- Higher level
- More concise
- More robust



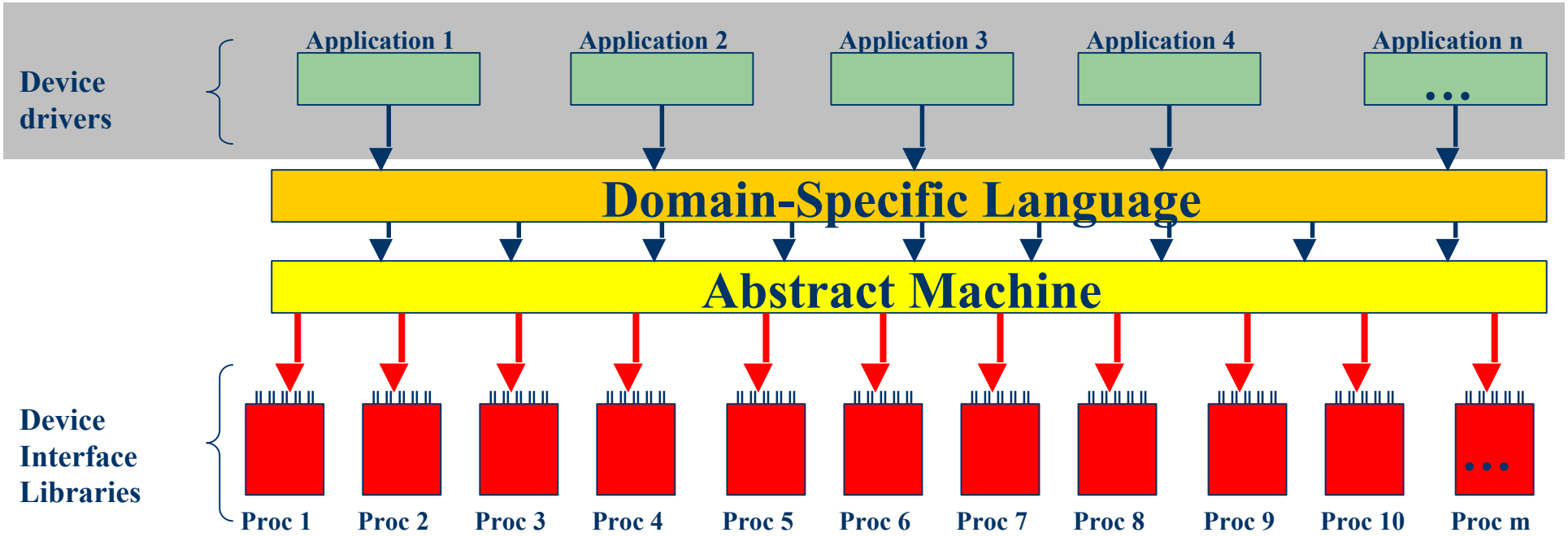
Program Family: Applications

- General-purpose language
  - General purpose glue for device communications
  - Checks subject to programmer's rigor
  - Repetitive communication sequences

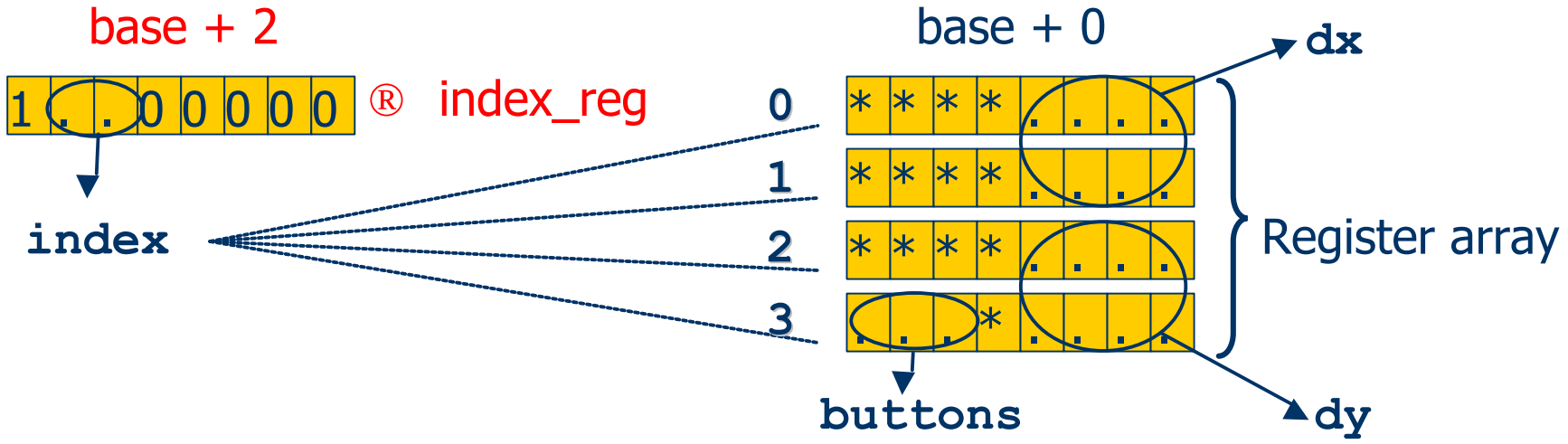


Key concept:
Layered Device Interface

- Port
  - Communication point
  - Different media: I/O, memory mapped
- Register
  - Repository of data in the device
  - indexed / paged, read / write, size, mask, …
- Variable
  - Collection of register fragments
  - Semantic value

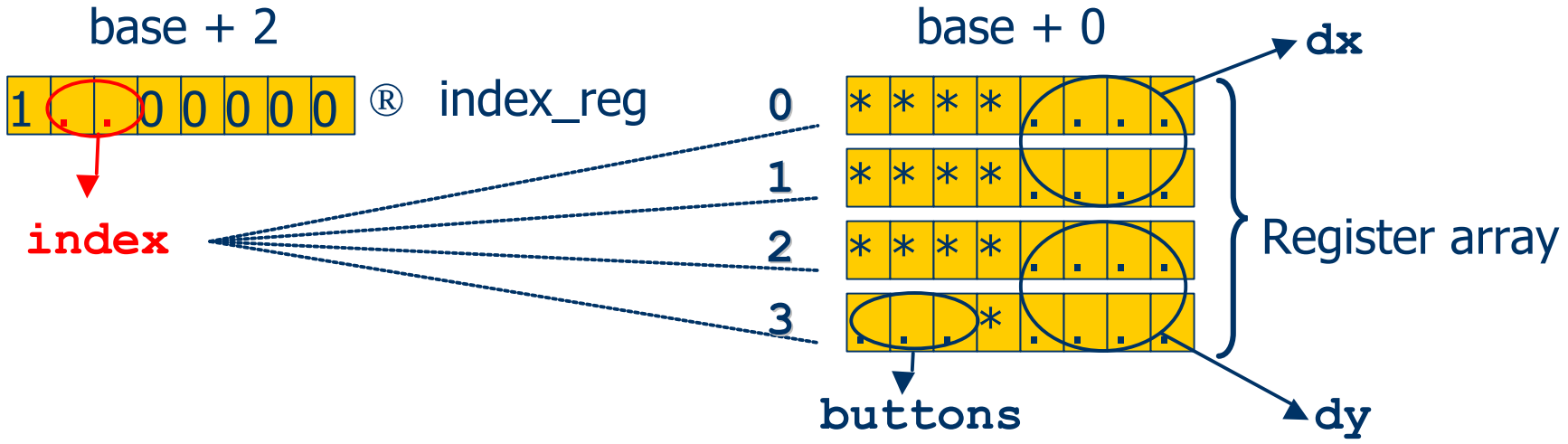


Direct Access to Register



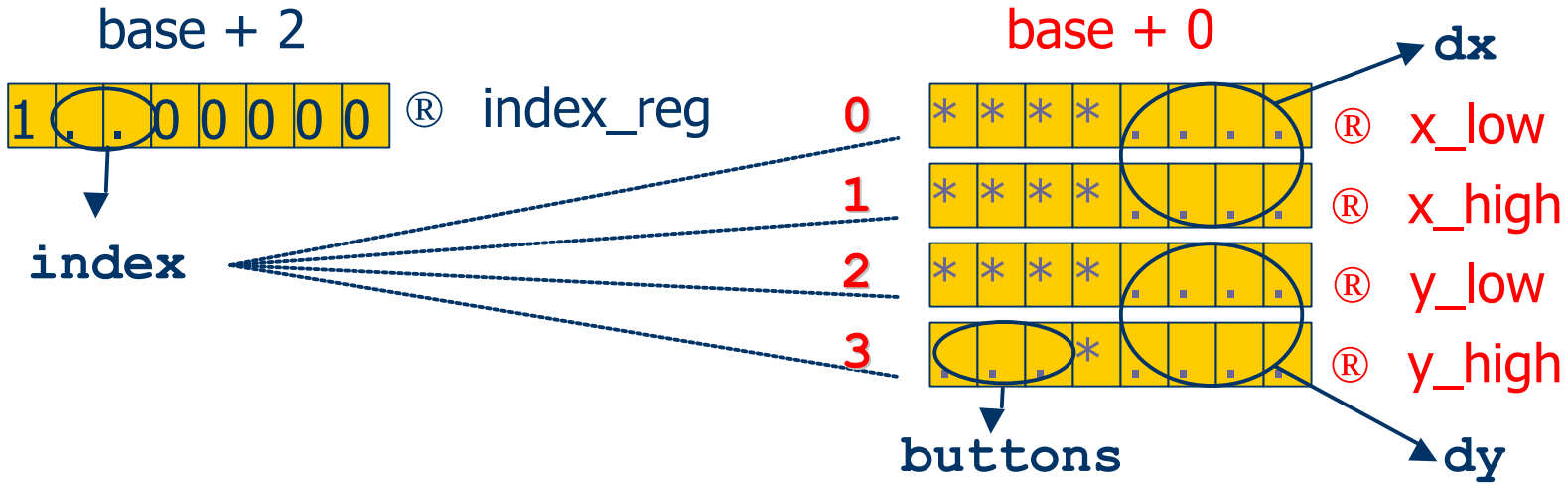

```
...
register index_reg = write base@2, mask '1..00000': bit[8];
```

Private Variable



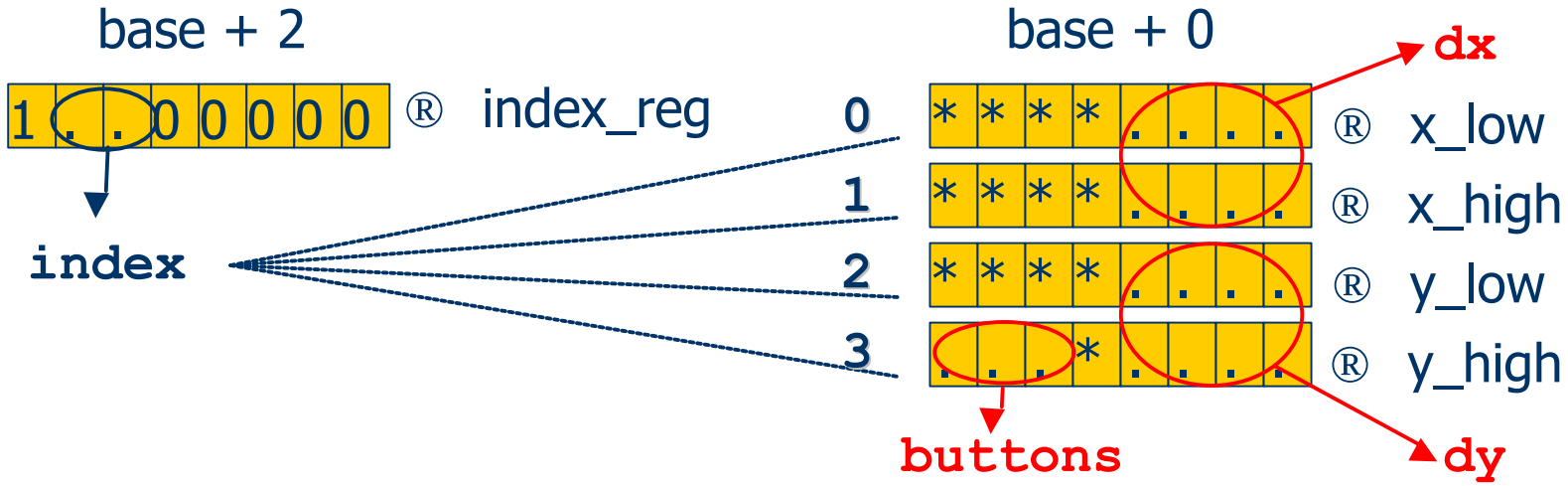

```
...
register index_reg = write base@2, mask '1..00000' : bit[8];
private variable index = index_reg[6..5] : int(2);
```

Indexed Registers




```
...
register index_reg = write base@2, mask '1..00000' : bit[8];
private variable index = index_reg[6..5] : int(2);
...
register x_low  = read base@0, pre {index = 0}, mask '****....' : bit[8];
register x_high = read base@0, pre {index = 1}, mask '****....' : bit[8];
...
```

Variables




```
...
register index_reg = write base@2, mask '1..00000' : bit[8];
private variable index = index_reg[6..5] : int(2);
...
register x_low  = read base@0, pre {index = 0}, mask '****....' : bit[8];
register x_high = read base@0, pre {index = 1}, mask '****....' : bit[8];
...
variable dx = x_high[3..0] # x_low[3..0], volatile : signed int(8);
variable dy = y_high[3..0] # y_low[3..0], volatile : signed int(8);
```

Comparison

```
outb(MSE_READ_X_LOW, MSE_CONTROL_PORT);
dx = (inb(MSE_DATA_PORT) & 0xf);
outb(MSE_READ_X_HIGH, MSE_CONTROL_PORT);
dx |= (inb(MSE_DATA_PORT) & 0xf) << 4;
outb(MSE_READ_Y_LOW, MSE_CONTROL_PORT );
dy = (inb(MSE_DATA_PORT) & 0xf);
outb(MSE_READ_Y_HIGH, MSE_CONTROL_PORT);
buttons = inb(MSE_DATA_PORT);
dy |= (buttons & 0xf) << 4;
```

Existing code

Devil

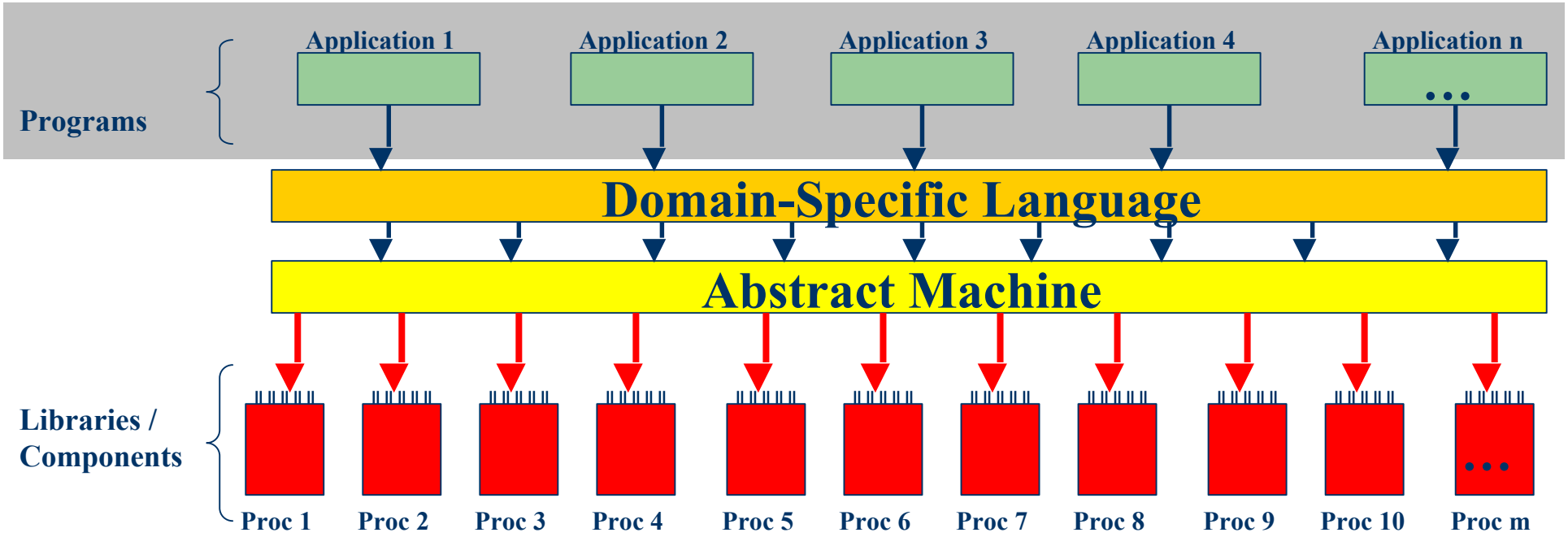
```
variable dx = x_high[3..0] # x_low[3..0],volatile
    : signed int(8);
variable dy = y_high[3..0] # y_low[3..0],volatile
    : signed int(8);
```

Applications:
Need for Domain-Specific Language

Devil language

```
variable dx = x_high[3..0] # x_low[3..0], volatile
     : signed int(8);
variable dy = y_high[3..0] # y_low[3..0], volatile
     : signed int(8);
```

- Re-use
- Expertise
- Conciseness
- Verification
- Productivity
- Performance



Two Final Points

- Assessing a DSL
- DSL variations

Assessing a DSL

- Existing program family
  - (Almost) objective comparisons:
    - Conciseness (Devil: -, Plan-P: 3 × smaller)
    - Robustness (Devil: specification consistency, interface usage)
    - Performance (Devil: equivalent to C)
  - (Likely) subjective comparisons:

    Readability, maintainability, usability

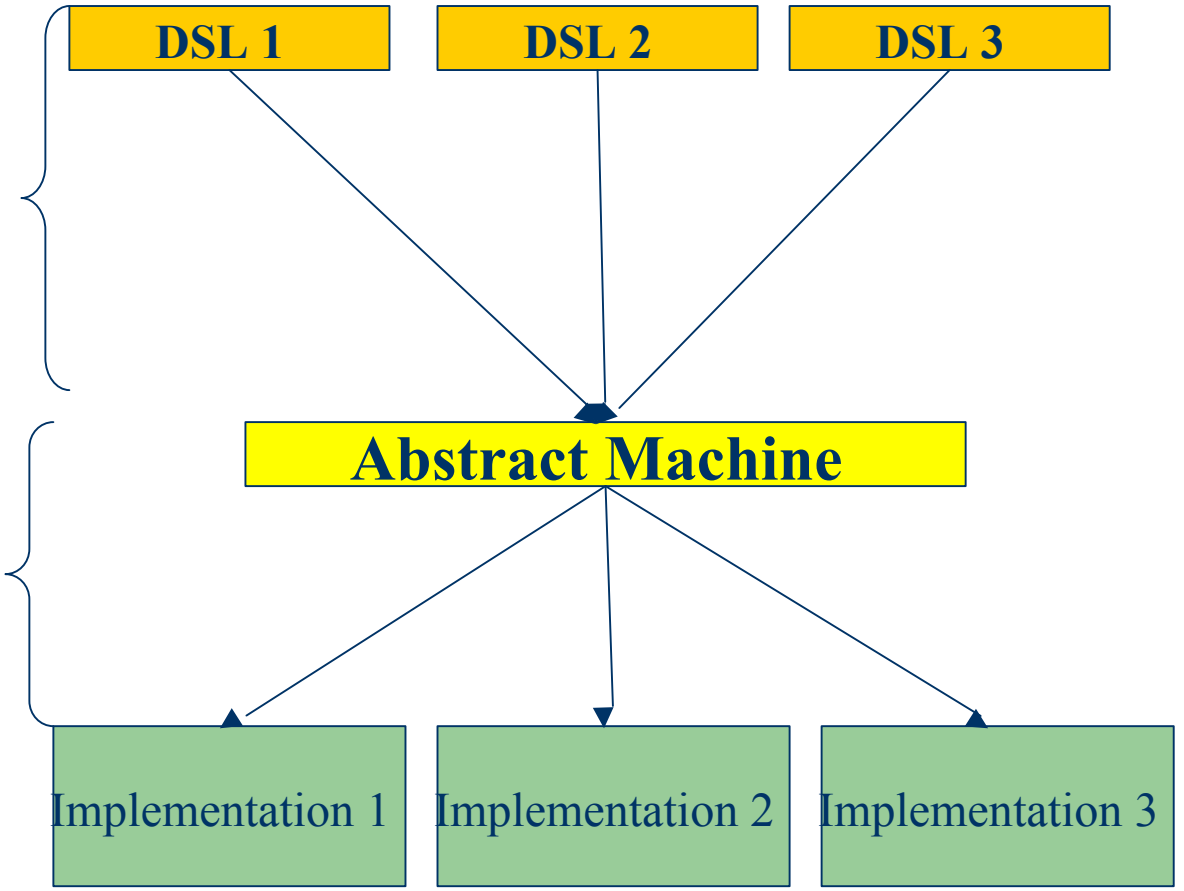
DSL Variations

- DSLs:

  Interfaces to the AM
  - Expressiveness
  - Safety

- AM:

  Interface to *libraries*
  - Implementations
  - Non-functional properties





Flexible, Safe, and Efficient Dynamic Generation of XML

Claus Brabrand

LaBRI/INRIA

Abstract

Increasingly, HTML/XML documents on the Web are dynamically generated by embedded server-side scripting languages, such as PHP, ASP, and JSP. However, this has five serious limitations: it forces linear document construction, intermixes designer/programmer aspects, yields implicit control-flow, prevents static XML validation, and precludes caching.

This talk presents a *flexible*, *safe*, and *efficient* language for dynamically generating XML documents that solves the above problems. We also show how to augment XHTML documents with form-field validation through declarative specifications.

More information:

The <bigwig> Project:

http://www.brics.dk/bigwig/

Outline

- Introduction
- Dynamic Documents
- Type Safety
- HTML Validation
- Caching
- Form-Field Validation
- Conclusion

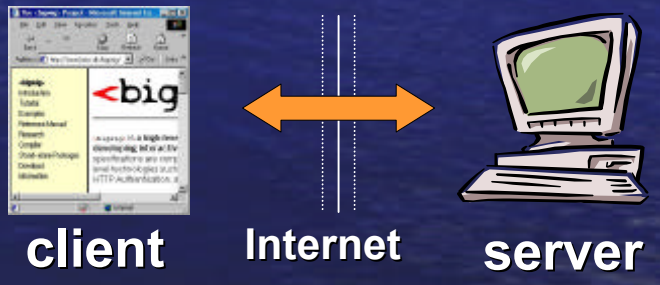
Outline

- Introduction
- Dynamic Documents
- Type Safety
- HTML Validation
- Caching
- Form-Field Validation
- Conclusion

Interactive (Form-Based) Web Services

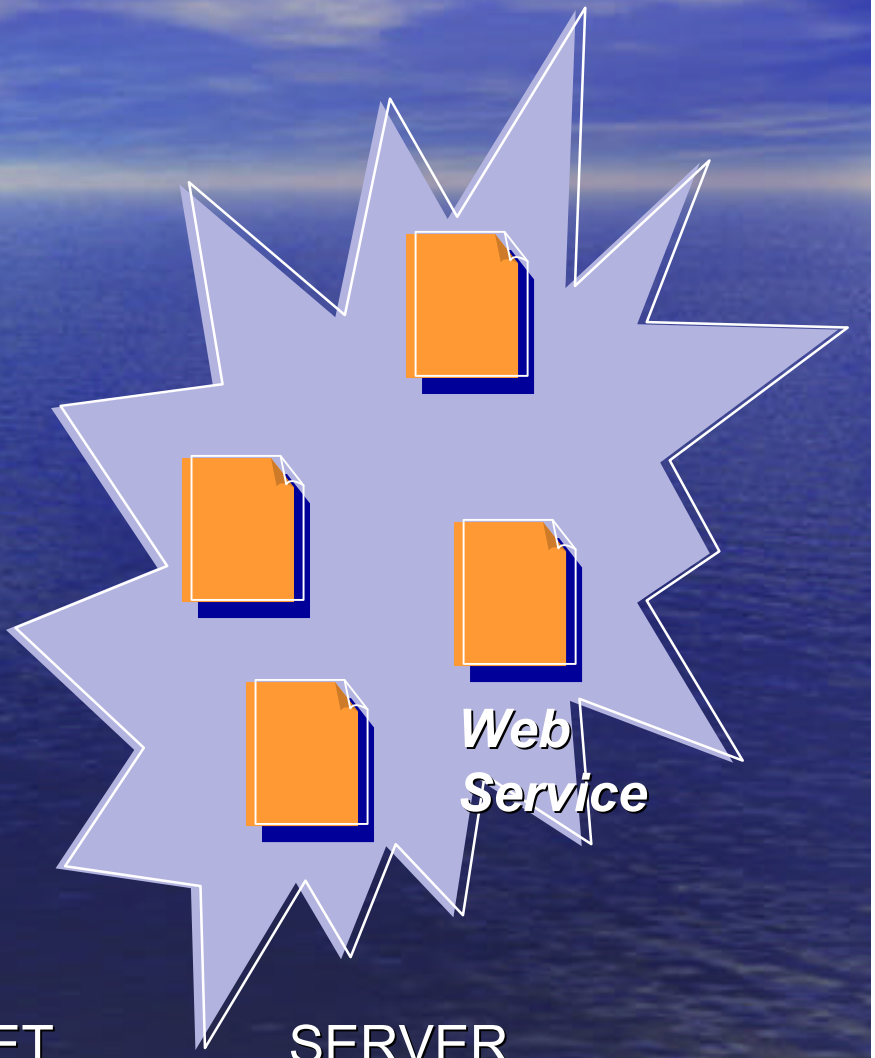
”Web servers on which clients can initiate sessions that involve several exchanges of information mediated by HTML forms”.

- Approaches:
  - Script-Centered
  - Page-Centered
  - Session-Centered



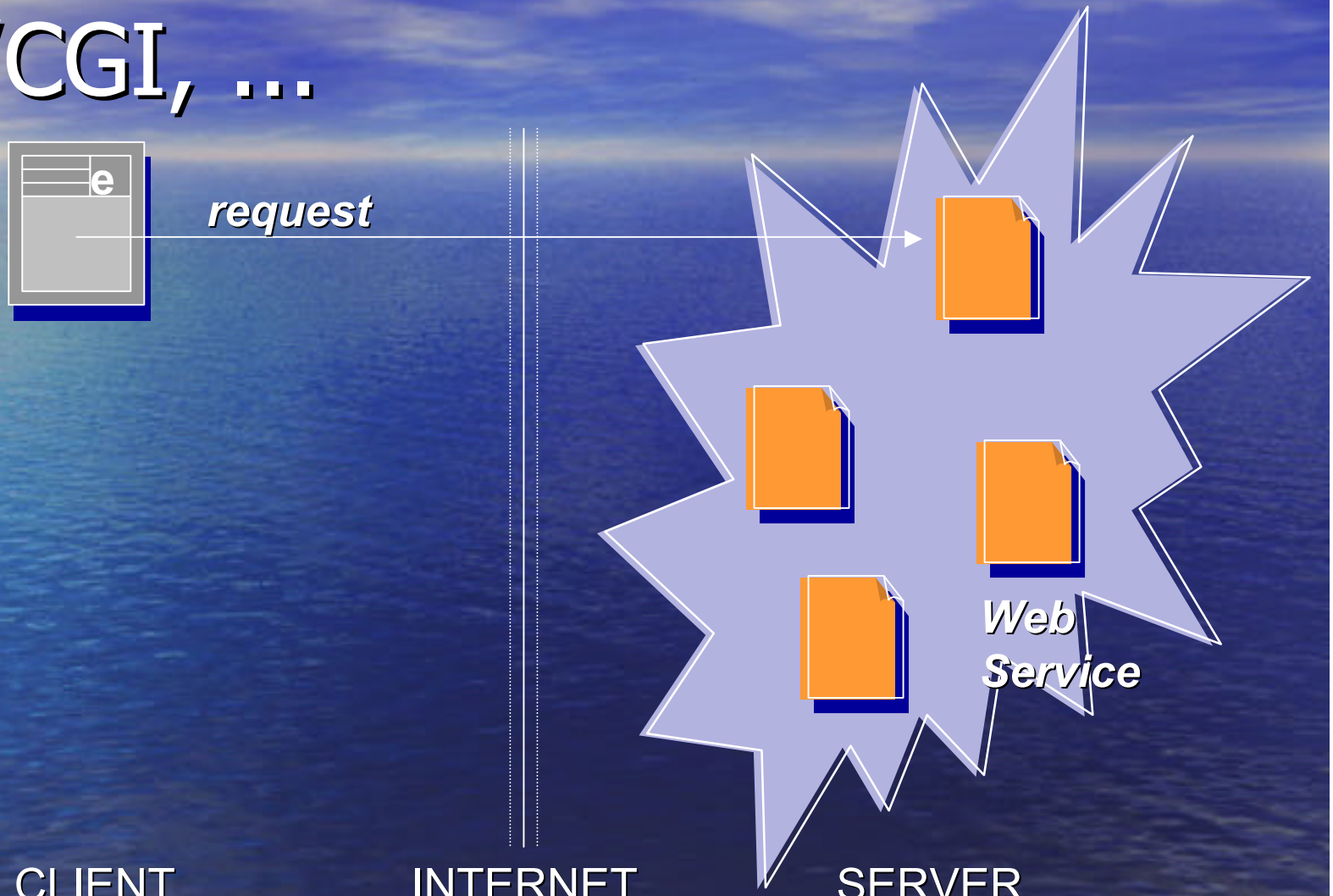
Script-Centered: Perl/CGI, ...



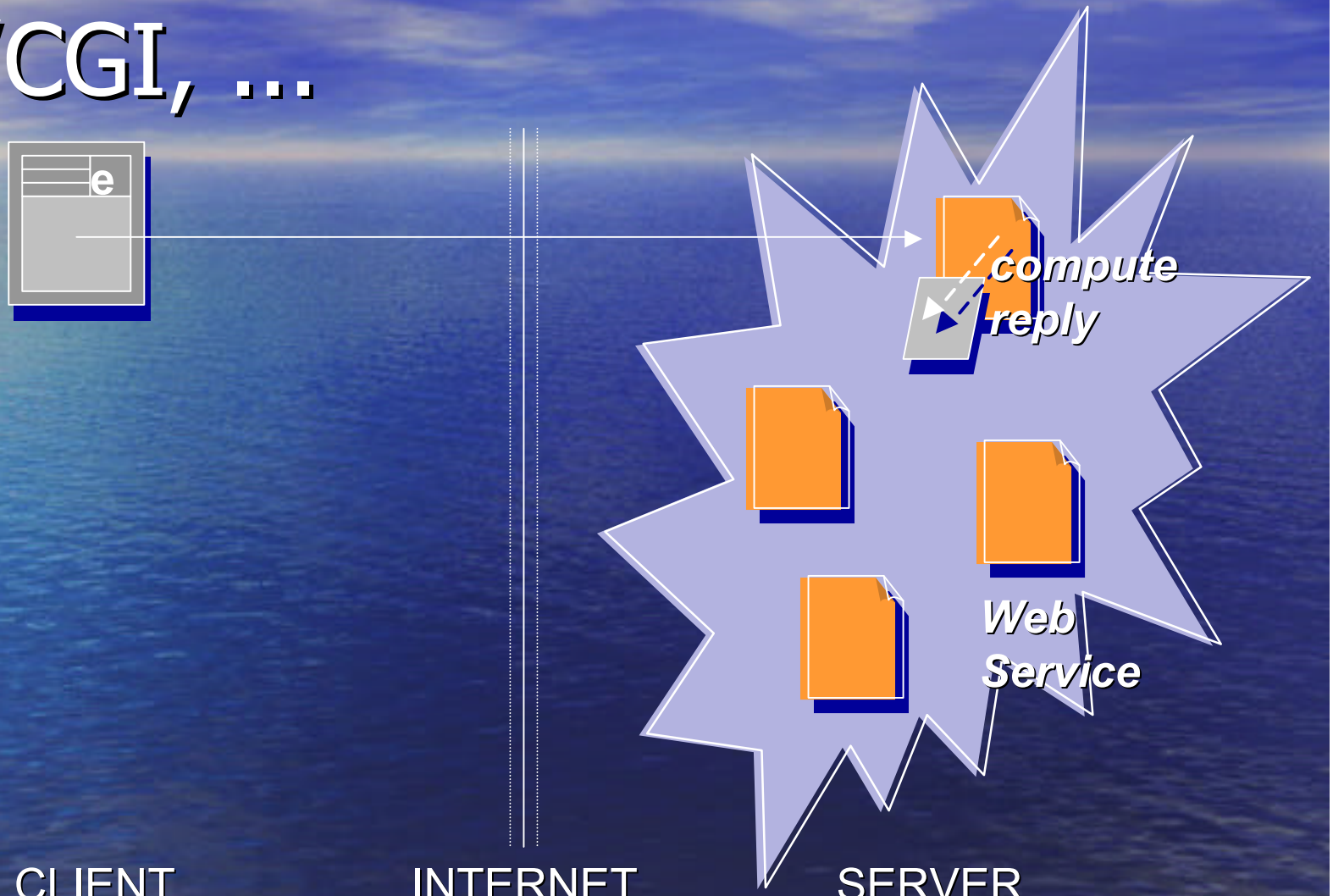


CLIENT INTERNET SERVER

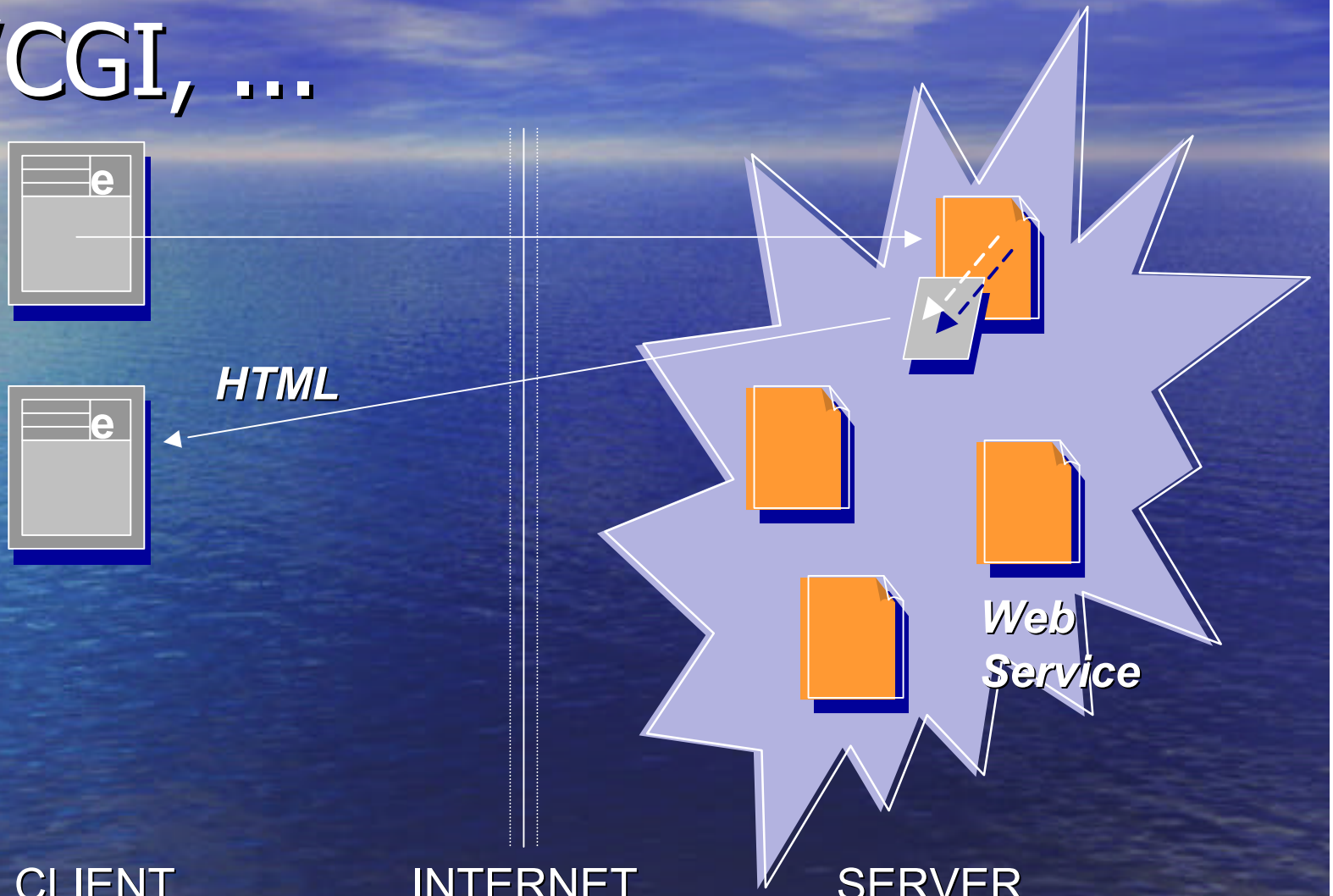
Script-Centered: Perl/CGI, ...



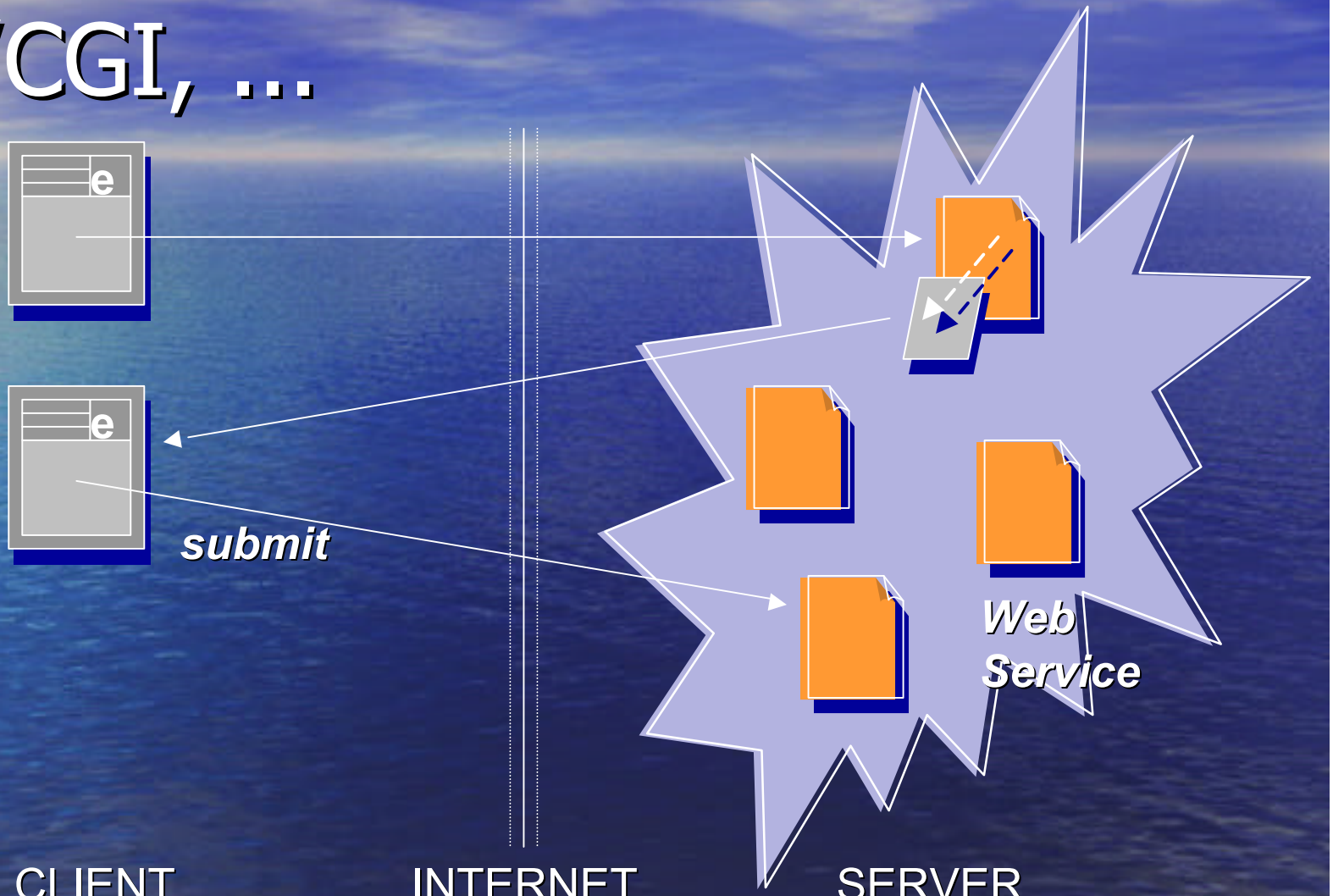
Script-Centered: Perl/CGI, ...



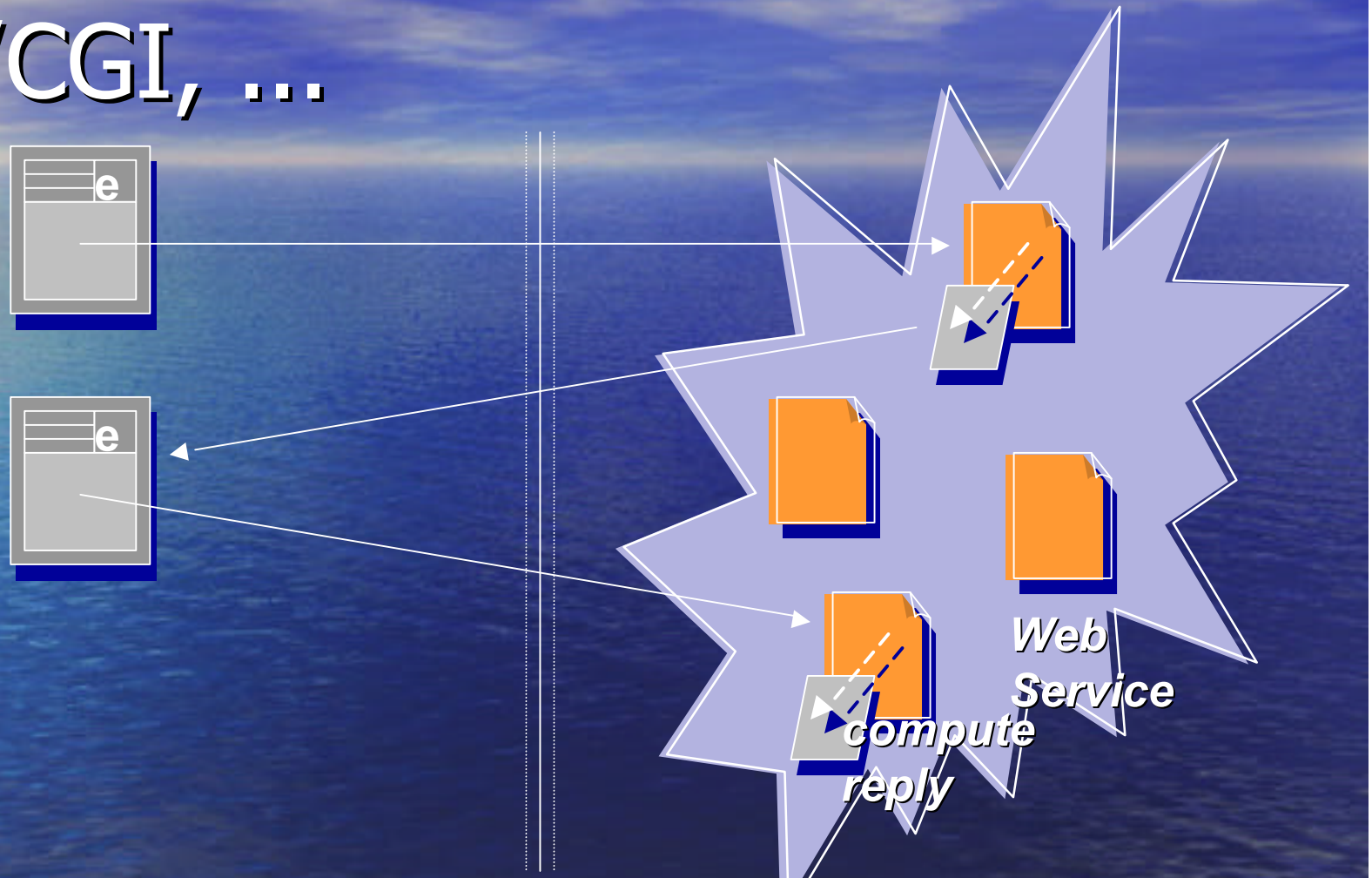
Script-Centered: Perl/CGI, ...



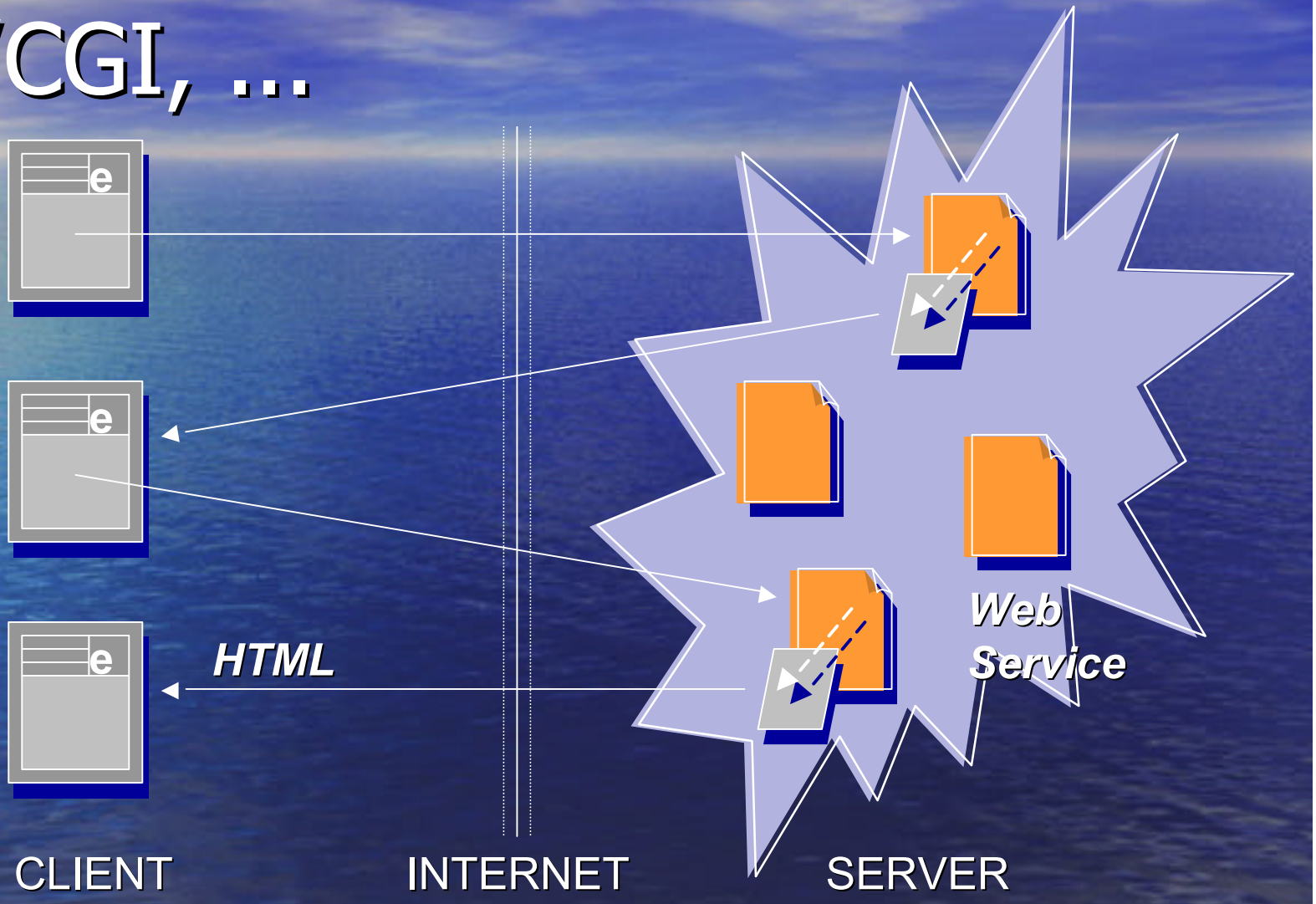
Script-Centered: Perl/CGI, ...



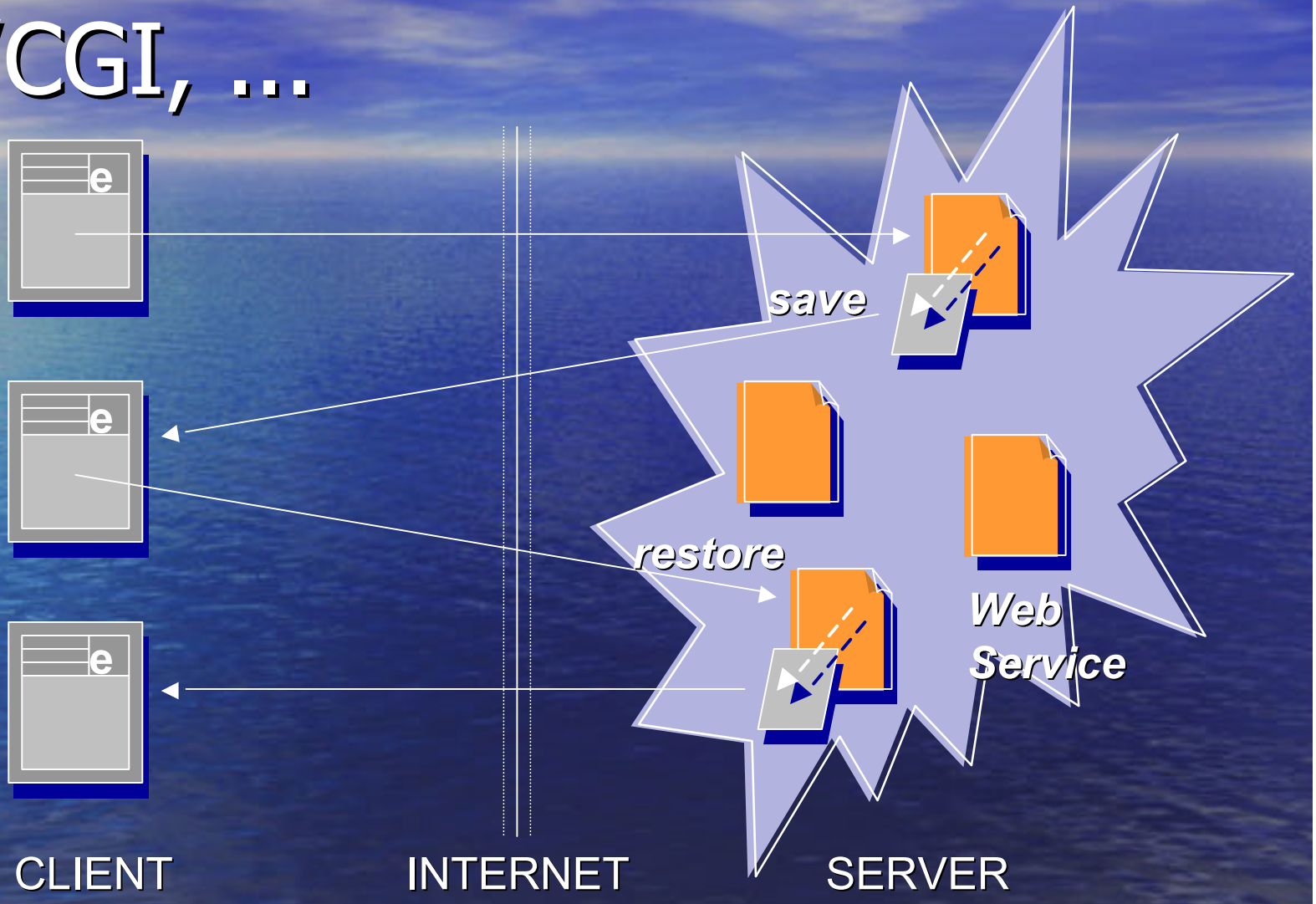
Script-Centered: Perl/CGI, ...



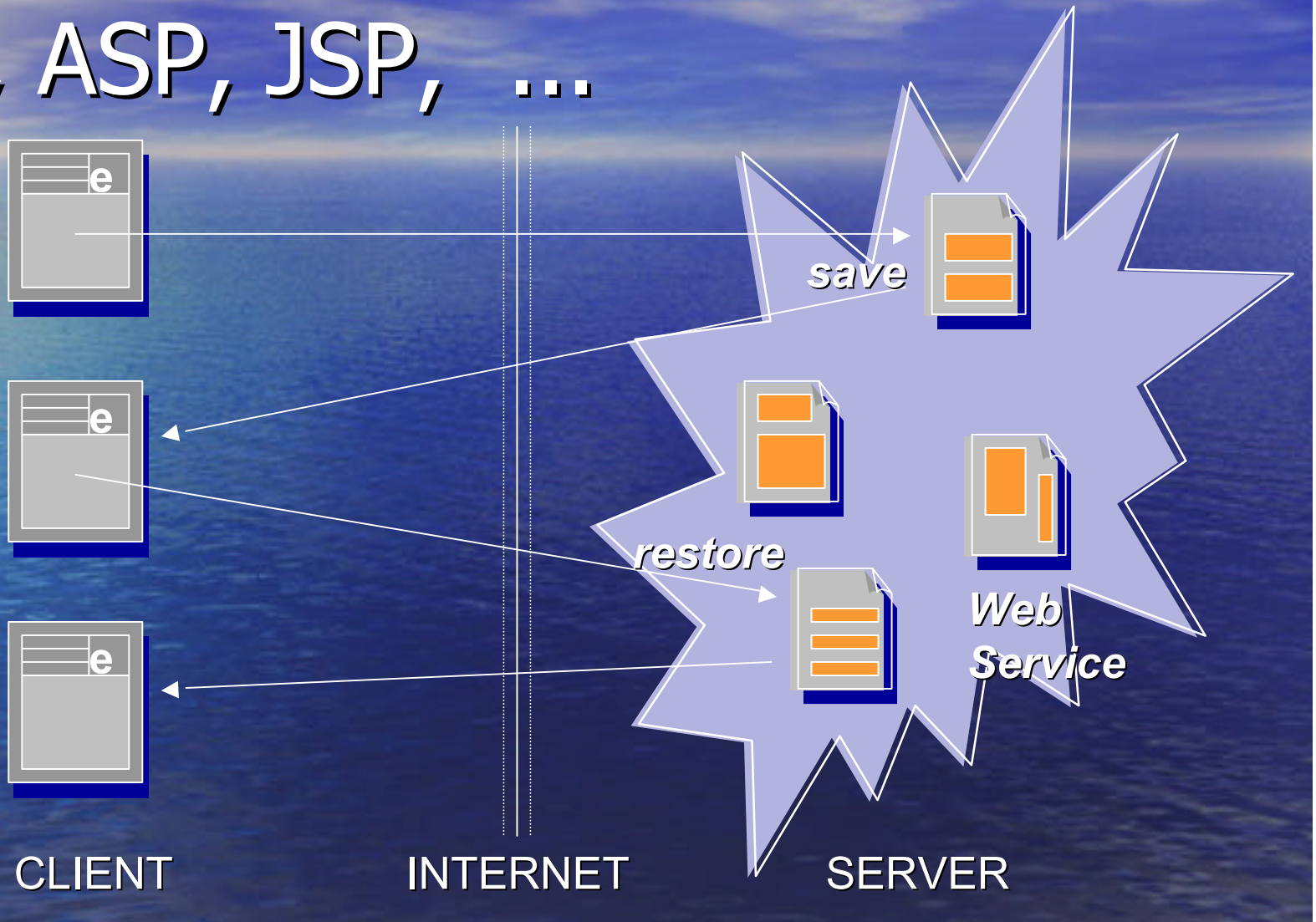
Script-Centered: Perl/CGI, ...



Script-Centered: Perl/CGI, ...



Page-Centered: PHP, ASP, JSP, ...



Page-Centered

"Service code embedded in tags and interpreted by specialized Web server"

- Increased level of abstraction:
  - CGI protocol details abstracted away
- Easy to add dynamics to static pages:
  - `<html><body>Time: <%= Now() %></body></html>`
- Scalability:
  - Specialized Web server

Script- vs. Page-Centered

- As the service complexity increases:
  - Page-Centered $\rightarrow$ Script-Centered
    - A lot of HTML is generated by script elements
- Interesting duality:
  - Script-centered:
    - Default: programming, Escape: printing (`print`)
  - Page-centered:
    - Default: printing, Escape: programming (<%...%>)

Both: Fundamental Drawbacks!

- Interpretation-based:
  - Errors at runtime
  - (Efficiency)



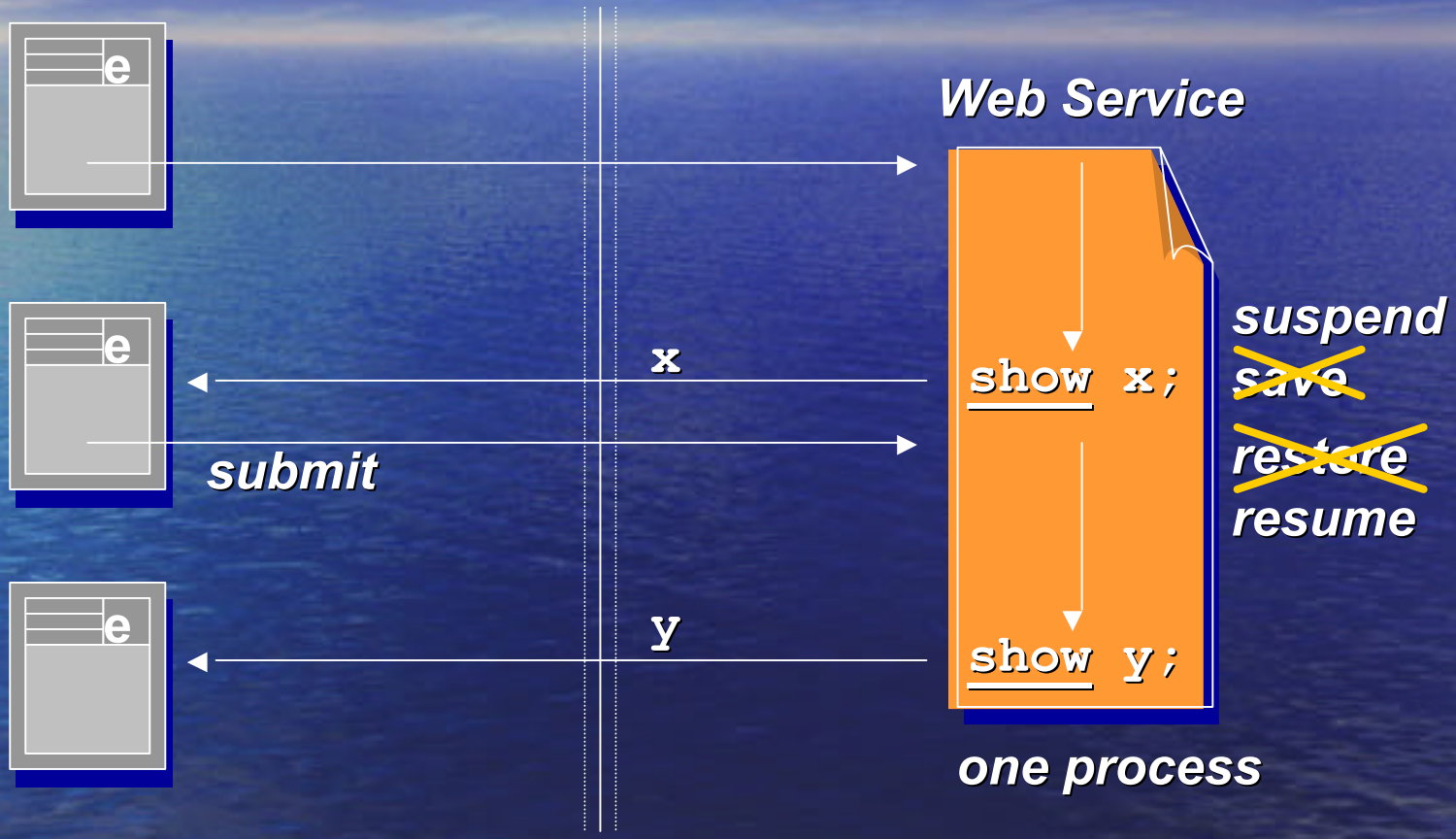
- A service = A collection of scripts/pages!
  - Implicit control-flow:
    - No interaction correspondence:



Language Requirements!

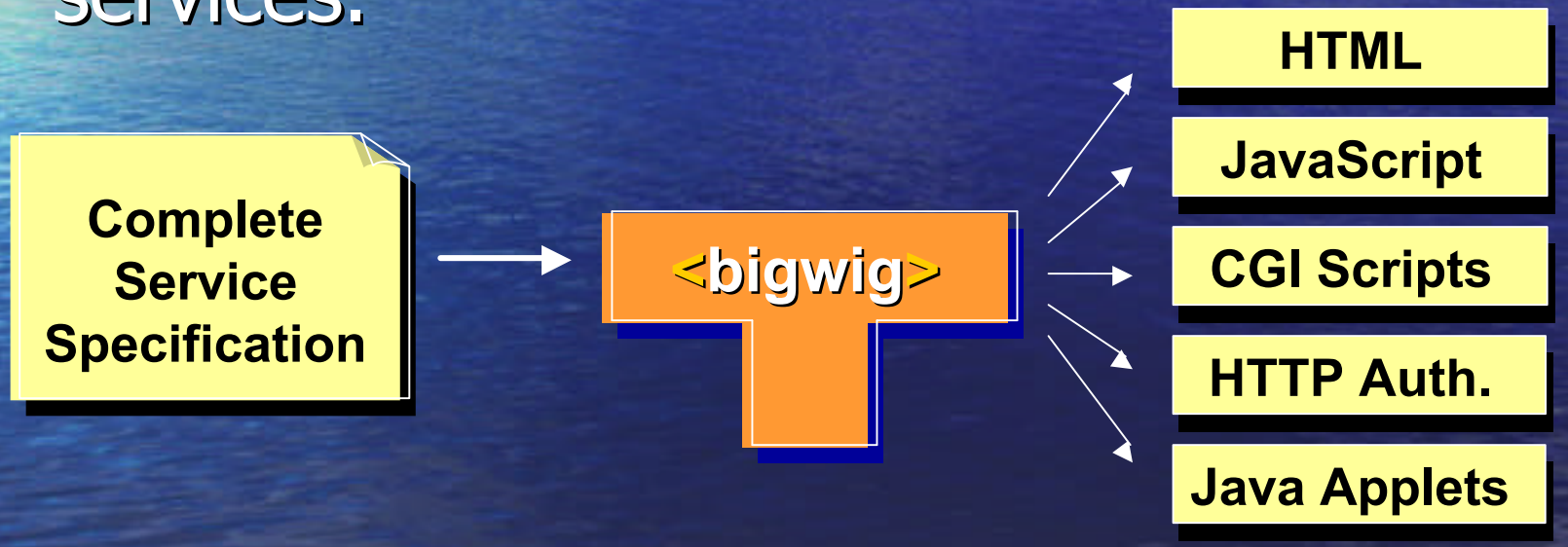
- Compilation-based:
  - Errors at compile-time
  - (Efficiency)
- A complete service specification
  - Explicit control-flow:
    - Statically check interaction correspondence

Session-Centered: Mawl and <bigwig>!



What is <bigwig>?

- A high-level domain-specific programming language for developing interactive Web services.



A collection of Domain Specific Languages

- C-like core language with
  - Session-based runtime system
  - Dynamic documents
  - Form-field validation
  - Relational database
  - Concurrency control
  - Semantic security
  - Cryptographic security
  - Syntactic-level macros

A collection of Domain Specific Languages

- C-like core language with
  - Session-based runtime system
  - Dynamic documents
  - Form-field validation
  - Relational database
  - Concurrency control
  - Semantic security
  - Cryptographic security
  - Syntactic-level macros

Outline

- Introduction
- Dynamic Documents
- Type Safety
- HTML Validation
- Caching
- Form-Field Validation
- Conclusion

Outline

- Introduction
- Dynamic Documents
- Type Safety
- HTML Validation
- Caching
- Form-Field Validation
- Conclusion

Dynamically Generated XML: Current Problems

- Traditionally:
  - `print(...)` in Perl/CGI, ...
  - `<% print(...) %>` in PHP, ASP, JSP, ...
- Problems:
  - Forces linear document construction
  - Intermixes programmer / designer aspects
  - No interaction correspondence
  - No static XML validation
  - No common fragments caching

Our Solution: HTML documents in <bigwig>

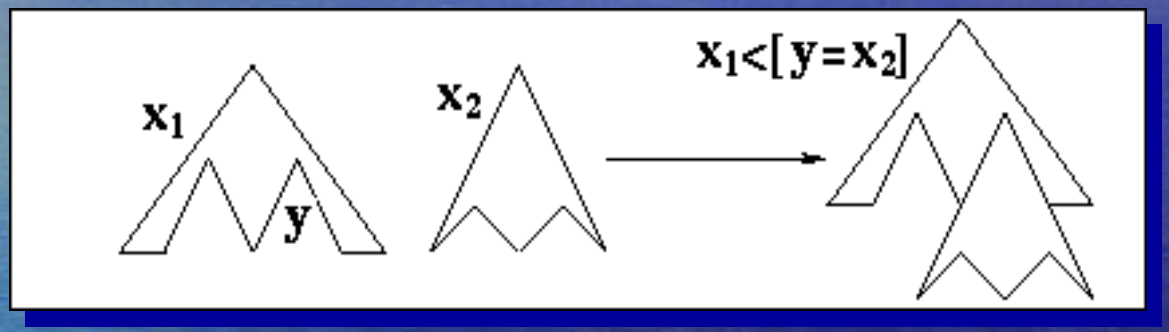
- Templates (1st class, higher-order values):
  - XML fragments with named gaps:

```
<html>
  Hello <[what]>!
</html>
```

- Operations:
  - plug: for document construction
  - show: for client interaction

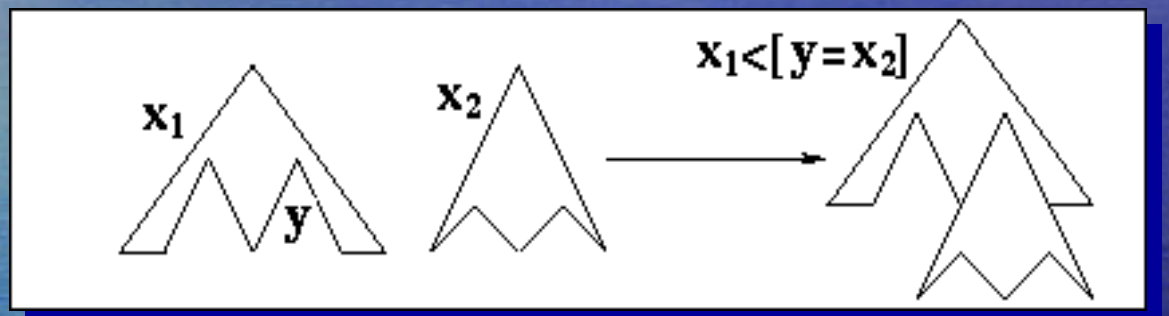
Plug

- Plug: exp < [id = exp]



Plug

- Plug: exp <[id = exp]



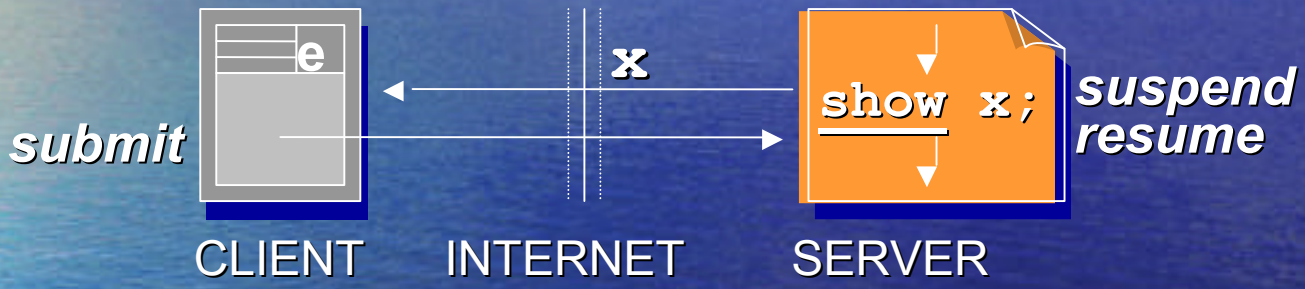
- Example:

```
html hello = <html>Hello <[what]>!</html>;
html world = <html><b>World</b></html>;

html h = hello <[what = world];
...;
```

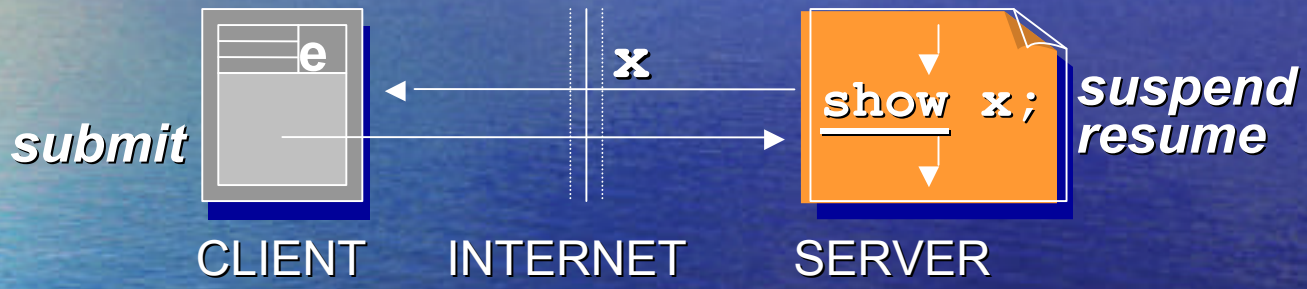
Show

- Show: **show** exp ;



Show

- Show: `show` exp `;`



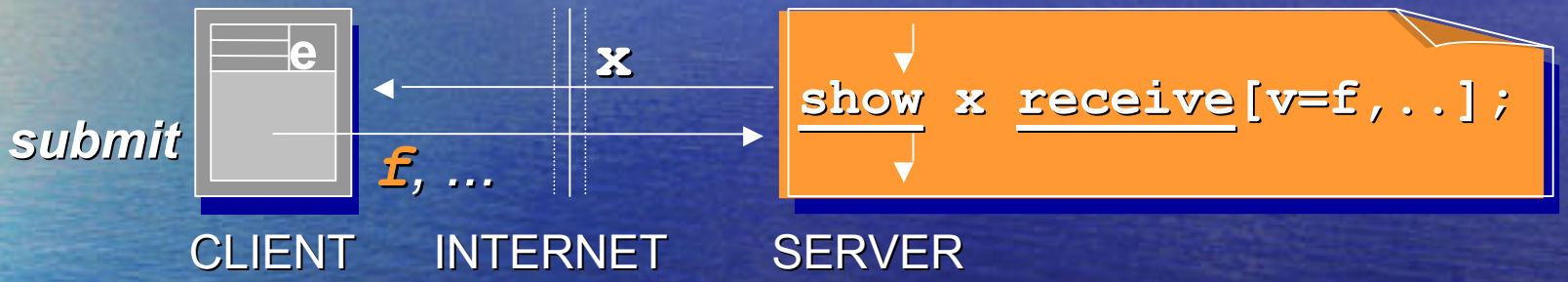
- Example:

```
html hello = <html>Hello <[what]>!</html>;
html world = <html><b>World</b></html>;

html h = hello <[what = world];
show h;
```

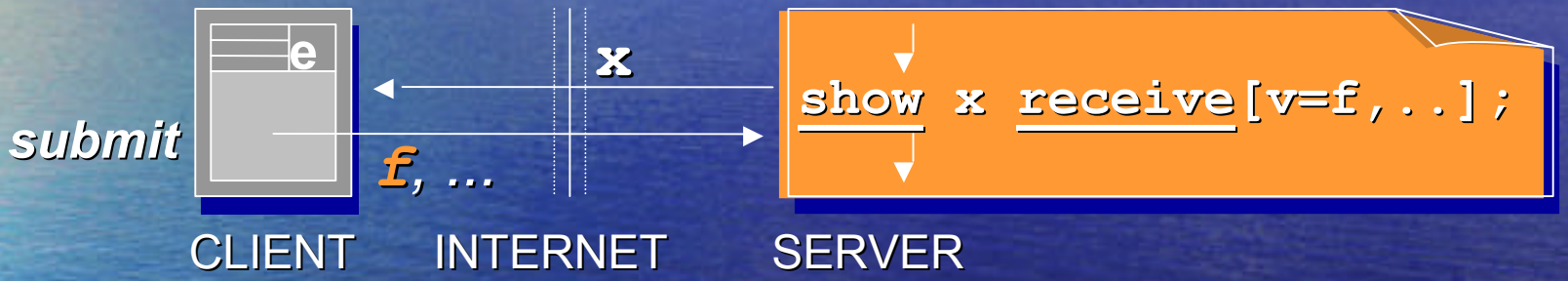
Show-Receive

- Show: `show` exp `receive[v=f, ...];`



Show-Receive

- Show: `show exp receive[v=f, ...];`



- Example:

```
string s;
html input = <html>
  Enter email: <input type="text" name="email">
</html>;

show input receive[s = email];
```

Example: "Welcome"

```
html greeting = <html>
  Hello <[who]>, welcome to <[what]>.
</html>;

html cover = <html>
  <head><title>Welcome</title></head>
  <body><[contents]></body>
</html>;

html h;
h = greeting <[who = "Stranger"];
h = h <[what = <html><b>BRICS</b></html>];
show cover <[contents = h];
```

Flexibility: PHP/ASP/JSP

- List of results (e.g. "search engine"):
  - One template with 10,20,30, or 40 hardwired server-side script elements:

...or...

  - One template with one big "generate-all" server-side script element:

≈ script-centered

```
<% ..1.. %>
<% ..2.. %>
   :
<% ..20.. %>
```

```
<% for
 i=1 to N do
 { ..i.. } %>
```

Flexibility: <bigwig>

- List of results (e.g. “search engine”):

```
<ul>
  <li>1
  <li>2
    :
  <li>N
</ul>
```

Flexibility: <bigwig>

- List of results (e.g. "search engine"):
  - Two templates: "`layout`" and "`entry`"...

```
<ul>
  <li>1
  <li>2
    :
  <li>N
</ul>
```

```
html layout = <html><ul><[items]></ul><html>;
html entry = <html><li><[num]><[items]></html>;
```

Flexibility: <bigwig>

```
<ul>
  <li>1
  <li>2
    :
  <li>N
</ul>
```

- List of results (e.g. "search engine"):
  - Two templates: "`layout`" and "`entry`"...

```
html layout = <html><ul><[items]></ul><html>;
html entry = <html><li><[num]><[items]></html>;
```

  - ...and one recursive function: "`f`"

```
html f(int n) {
  if (n == 0) return layout;
  return f(n-1) <[items = entry <[num = n]];
}
```

Flexibility: <bigwig>

```
<ul>
  <li>1
  <li>2
    :
  <li>27
</ul>
```

- List of results (e.g. "search engine"):
  - Two templates: "`layout`" and "`entry`"...

```
html layout = <html><ul><[items]></ul><html>;
html entry = <html><li><[num]><[items]></html>;
```

  - ...and one recursive function: "`f`"

```
html f(int n) {
  if (n == 0) return layout;
  return f(n-1) <[items = entry <[num = n]];
}
```

  - Example usage:

```
show f(27);
```

Flexibility: Separates Designer / Programmer Aspects

```
html layout = <html><ul><[items]></ul><html>;
html entry = <html><li><[num]><[items]></html>;
```

```
html f(int n) {
  if (n == 0) return layout;
  return f(n-1) <[items = entry <[num = n]];
}
```

Flexibility: Separates Designer / Programmer Aspects

~~~~
html layout = <html><ul><[items]></ul><html>;
html entry = <html><li><[num]><[items]></html>;
~~~~

```
html layout = <html><table><[items]></table><html>;
html entry = <html>
  <tr><td><[num]></td></tr>
  <[items]>
</html>;
```

```
html f(int n) {
  if (n == 0) return layout;
  return f(n-1) <[items = entry <[num = n]];
}
```

Problems?

- "DynDoc":
  - Templates with named gaps
  - Plug and show
- Problems:
  - Forces linear document construction
  - Intermixes programmer / designer aspects
  - No interaction correspondence
  - No static XML validation
  - No common fragments caching

Problems?

- "DynDoc":
  - Templates with named gaps
  - Plug and show
- Problems:
  - ~~Forces linear document construction~~
  - Intermixes programmer / designer aspects
  - No interaction correspondence
  - No static XML validation
  - No common fragments caching

Problems?

- "DynDoc":
  - Templates with named gaps
  - Plug and show
- Problems:
  - ~~Forces linear document construction~~
  - ~~Intermixes programmer / designer aspects~~
  - No interaction correspondence
  - No static XML validation
  - No common fragments caching

Problems?

- "DynDoc":
  - Templates with named gaps
  - Plug and show
- Problems:
  - ~~Forces linear document construction~~
  - ~~Intermixes programmer / designer aspects~~
  - No interaction correspondence
  - No static XML validation
  - No common fragments caching

Problems?



- Problems:
  - ~~Forces linear document construction~~
  - ~~Intermixes programmer / designer aspects~~
  - No interaction correspondence
  - No static XML validation
  - No common fragments caching

Outline

- Introduction
- Dynamic Documents
- Type Safety
- HTML Validation
- Caching
- Form-Field Validation
- Conclusion

Outline

- Introduction
- Dynamic Documents
- Type Safety
- HTML Validation
- Caching
- Form-Field Validation
- Conclusion

Type Safety (in terms of <bigwig>)

- Show/Receive correspondence:

```
html d = <html>
  Enter email: <input type="text" name="email">
<html>;

show d receive[s = age];
```

☹

- Gap presence:

```
html hello = <html>Hello <[what]>!<html>;
html world = <html><b>World</b></html>;
html h;

h = hello <[contents = world];
```

☹

Static Analysis!

Apply standard, ***data-flow analysis*** techniques, but with highly *domain-specific lattices*

Type Safety (in terms of <bigwig>)

- Show/Receive interaction correspondence:

```
html d = <html>
  Enter email: <input type="text" name="email">
<html>;

show d receive[s = age];   ☹
```

- Gap presence:

```
html hello = <html>Hello <[what]>!<html>;
html world = <html><b>World</b></html>;
html h;

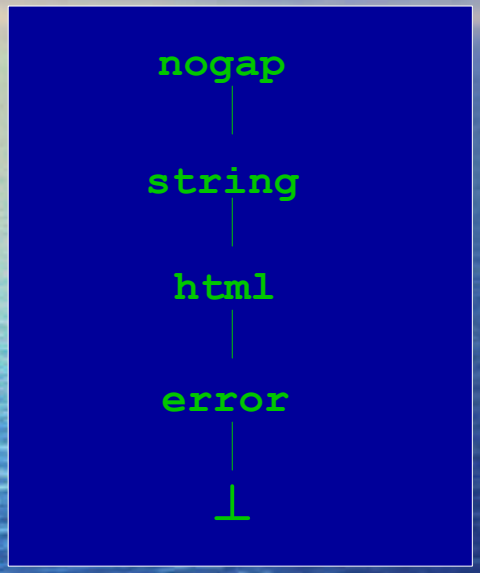
h = hello <[contents = world];   ☹
```

Static Analysis!

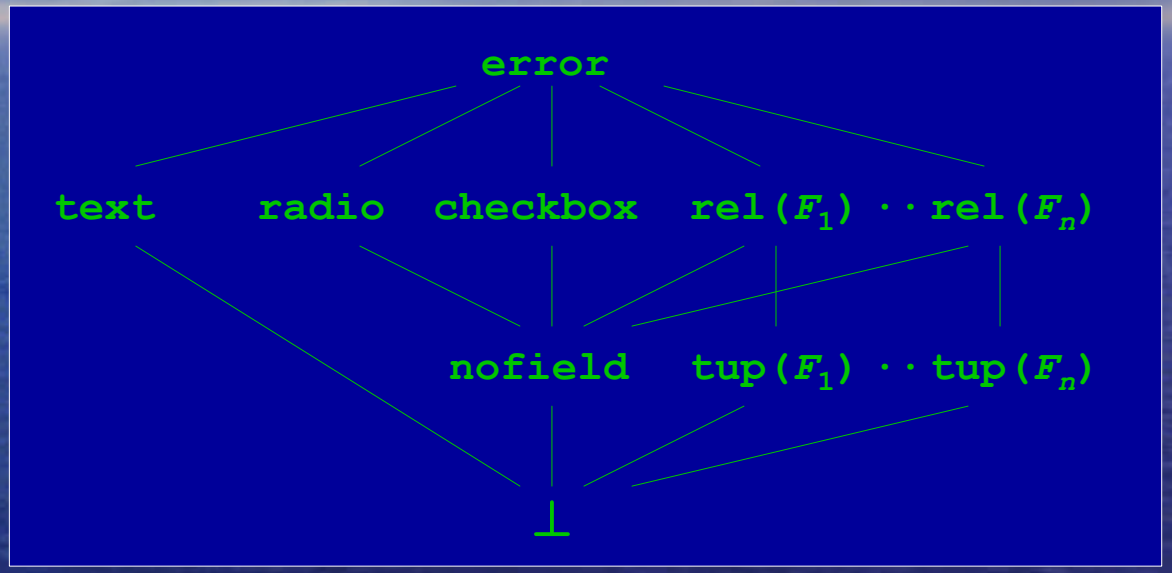
- Conservatively approximate
  for all HTML variables & program points:
  - Their gaps and fields (and their kinds)
    ~ all possible runtime values
- Forward data-flow analysis:
  - Control-flow graph (trivial for <bigwig>)
  - Finite lattice: Gap-and-Field
  - Monotone transfer functions: plug, assign, ...

Gap-and-Field Lattice

GKind



FKind



- An abstract document is:

  (GName → GKind) x (FName → FKind)

FName	®	FKind
GName	®	GKind
a	↦	html
g	↦	html
n	↦	nogap

Monotone Transfer Functions

- Plug: $x_1 <[g = x_2]$

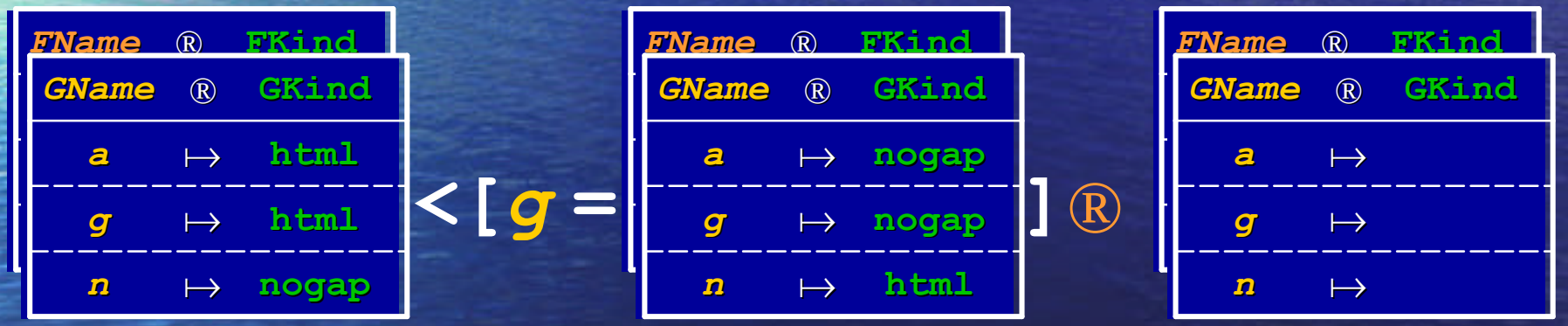
Monotone Transfer Functions

- Plug: $x_1 \langle [g = x_2] \rangle$



Monotone Transfer Functions

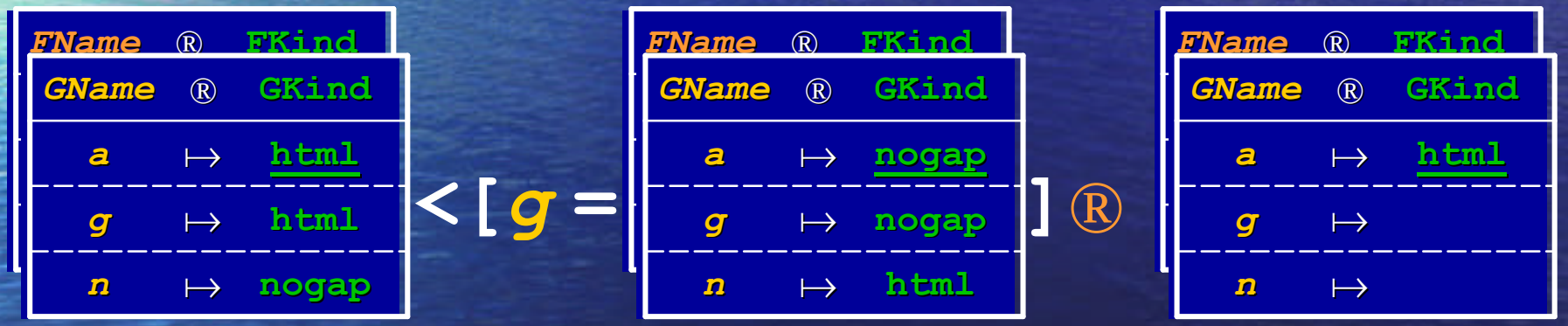
- Plug: $x_1 <[g = x_2]$



$Å_G : GKind \times GKind \rightarrow GKind$

Monotone Transfer Functions

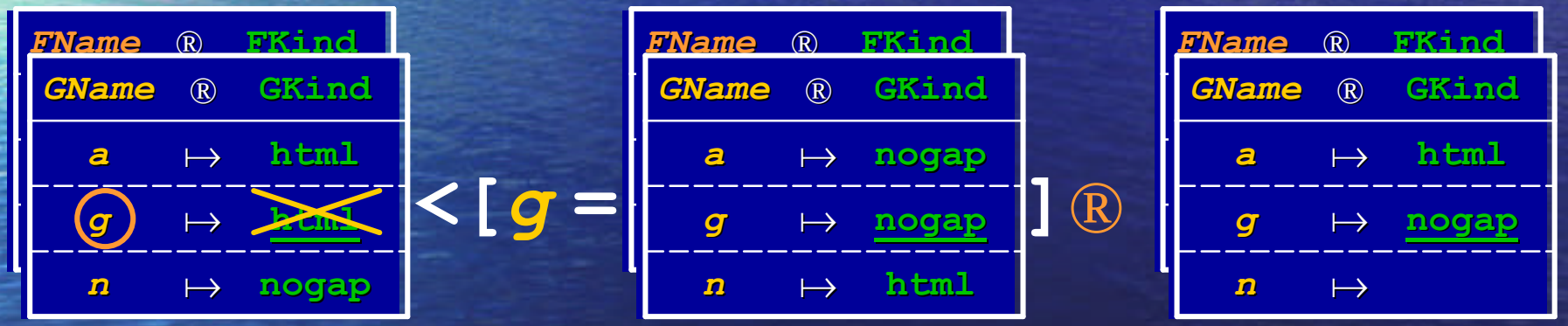
- Plug: $x_1 \langle [g = x_2]$



$\mathring{A}_G$: GKind x GKind → GKind

Monotone Transfer Functions

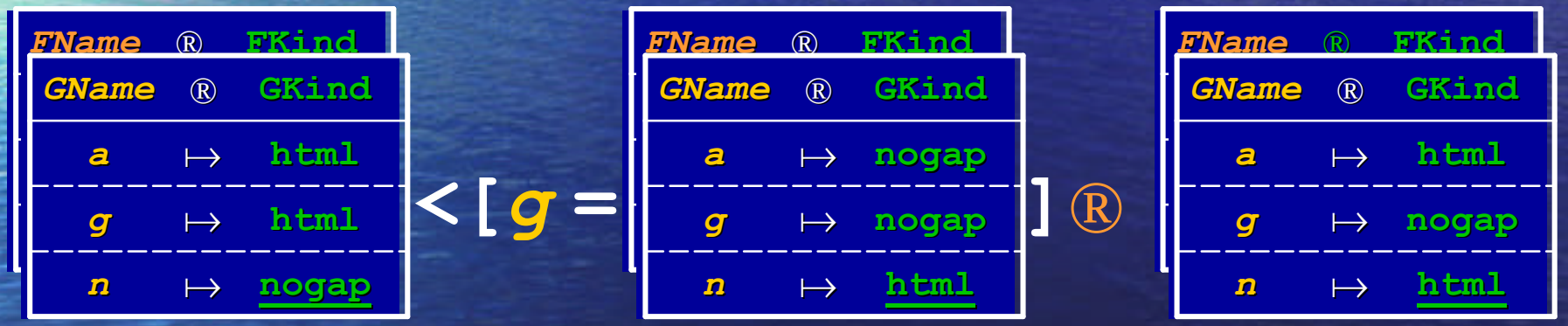
- Plug: $x_1 <[g = x_2]$



$Å_G$: GKind x GKind → GKind

Monotone Transfer Functions

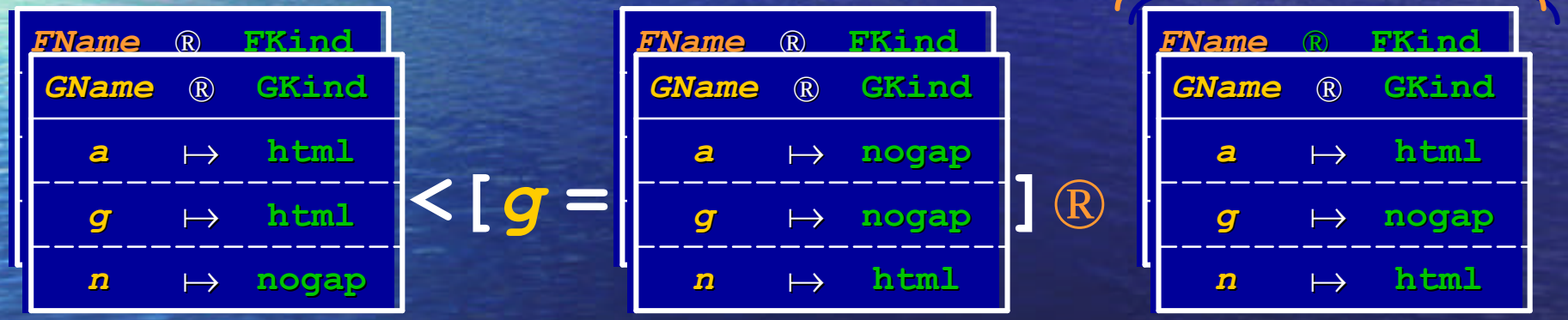
- Plug: $x_1 <[g = x_2]$



$Å_G : GKind \times GKind \rightarrow GKind$

Monotone Transfer Functions

- Plug: $x_1 <[g = x_2]$

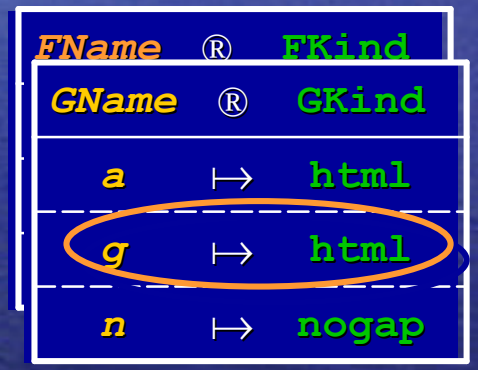


Then: Check Solution

- Traverse solution: intercept `errors`

Then: Check Solution

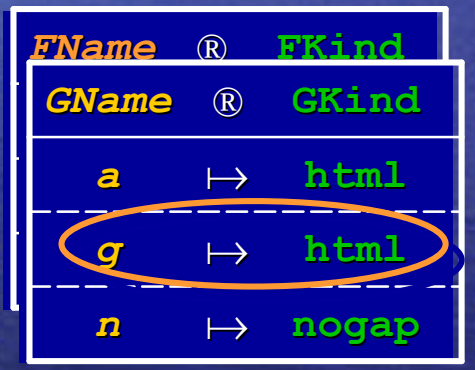
- Traverse solution: intercept **errors**
- $\forall$ $E_1 \prec [g = E_2]$



gap present

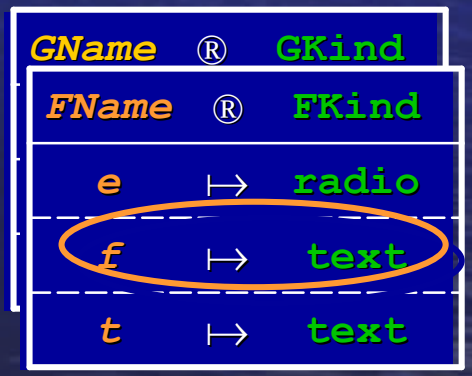
Then: Check Solution

- Traverse solution: intercept **errors**
- ∀ E_1 <[g = E_2]



gap present

- ∀ show E receive[v=f, …];



field present
type check: v ~ text

Type Safety?

- Show/Receive interaction correspondence:

```
html d = <html>
  Enter email: <input type="text" name="email">
<html>;

show d receive[s = age];
```

☹

- Gap presence:

```
html hello = <html>Hello <[what]>!<html>;
html world = <html><b>World</b></html>;
html h;

h = hello <[contents = world];
```

☹

Type Safety!

- Show/Receive interaction correspondence:

```
html d = <html>
  Enter email: <input type="text" name="email">
<html>;

show d receive[s = age];
```

```
*** receive.wig:5: input field 'email' not received
                   no such input field 'age'
```

- Gap presence:

```
html hello = <html>Hello <[what]>!<html>;
html world = <html><b>World</b></html>;
html h;

h = hello <[contents = world];
```

```
*** plug.wig:5: no such gap 'contents'
```

Problems?

- "DynDoc":
  - Templates with named gaps
  - Plug and show
- Problems:
  - ~~Forces linear document construction~~
  - ~~Intermixes programmer / designer aspects~~
  - No interaction correspondence
  - No static XML validation
  - No common fragments caching

Problems?

- "DynDoc":
  - Templates with named gaps
  - Plug and show
- Problems:
  - ~~Forces linear document construction~~
  - ~~Intermixes programmer / designer aspects~~
  - ~~No interaction correspondence~~
  - No static XML validation
  - No common fragments caching

Problems?

- "DynDoc":
  - Templates with named gaps
  - Plug and show
- Problems:
  - ~~Forces linear document construction~~
  - ~~Intermixes programmer / designer aspects~~
  - ~~No interaction correspondence~~
  - No static XML validation
  - No common fragments caching

Outline

- Introduction
- Dynamic Documents
- Type Safety
- HTML Validation
- Caching
- Form-Field Validation
- Conclusion

Outline

- Introduction
- Dynamic Documents
- Type Safety
- HTML Validation
- Caching
- Form-Field Validation
- Conclusion

Example: "Welcome"

```
html greeting = <html>
  Hello <[who]>, welcome to <[what]>.
</html>;

html cover = <html>
  <head><title>Welcome</title></head>
  <body><[contents]></body>
</html>;

html h;
h = greeting <[who = "Stranger"];
h = h <[what = <html><b>BRICS</b></html>];
show cover <[contents = h];
```

Example: "Welcome"

```
html greeting = <html>
  Hello <[who]>, welcome to <[what]>.
</html>;

html cover = <html>
  <head><title>Welcome</title></head>
  <body><[contents]></body>
</html>;

html h;
h = greeting <[who = "Stranger"];
h = h <[what = <html><b>BRICS</b></html>];
show cover <[contents = h];
```

Valid HTML !?

Static Analysis (again)!

- Conservatively approximate
  for all HTML variables & program points:
  - A summary graph ~ all possible runtime values
- Forward data-flow analysis:
  - Control-flow graph: (trivial for <bigwig>)
  - Finite lattice: summary graphs
  - Monotone transfer functions: plug, assign, ...

Summary Graph

```
html greeting = ...;

html cover = ...;

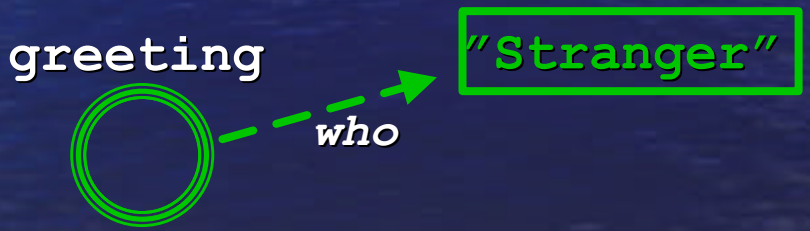
html h;
h = greeting <[who = "Stranger"];
h = h <[what = <html><b>BRICS</b></html>];
show cover <[contents = h];
```

Summary Graph

```
html greeting = ...;

html cover = ...;

html h;
h = greeting <[who = "Stranger"];
h = h <[what = <html><b>BRICS</b></html>];
show cover <[contents = h];
```

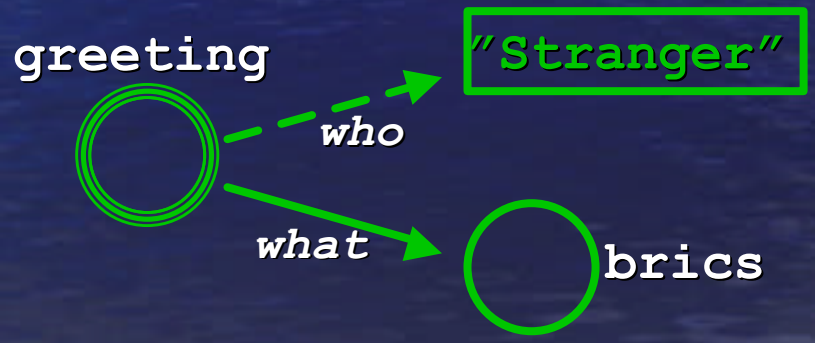


Summary Graph

```
html greeting = ...;

html cover = ...;

html h;
h = greeting <[who = "Stranger"];
h = h <[what = <html><b>BRICS</b></html>];
show cover <[contents = h];
```

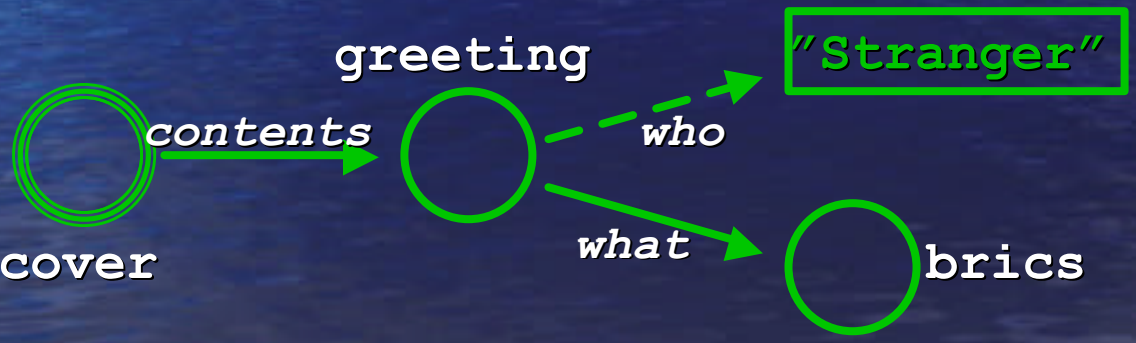


Summary Graph

```
html greeting = ...;

html cover = ...;

html h;
h = greeting <[who = "Stranger"];
h = h <[what = <html><b>BRICS</b></html>];
show cover <[contents = h];
```

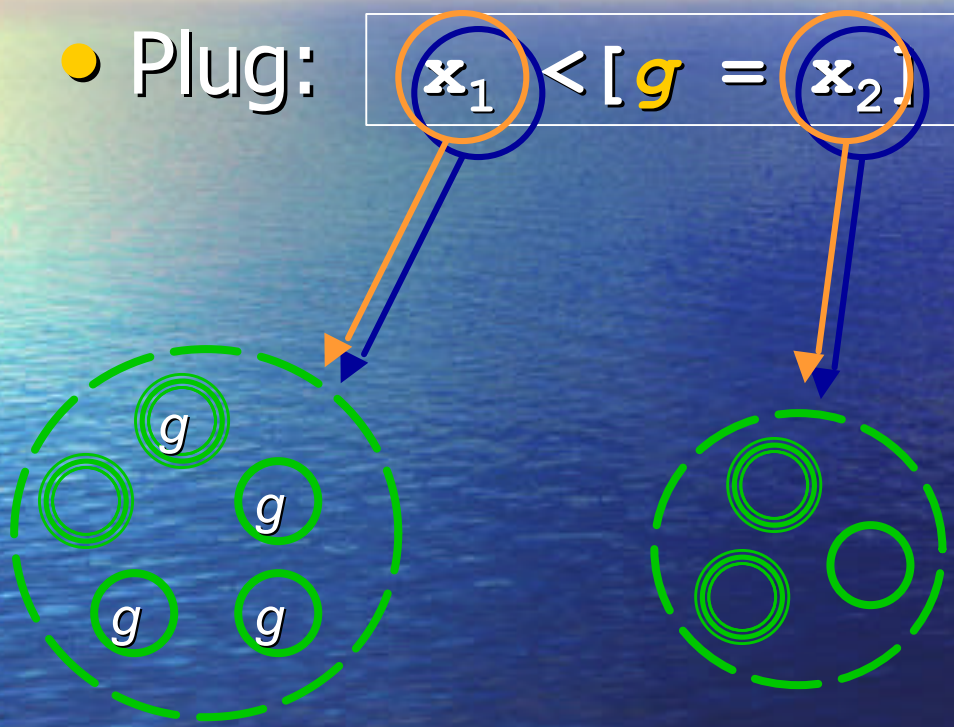


Monotone Transfer Function

- Plug: $x_1 <[g = x_2]$

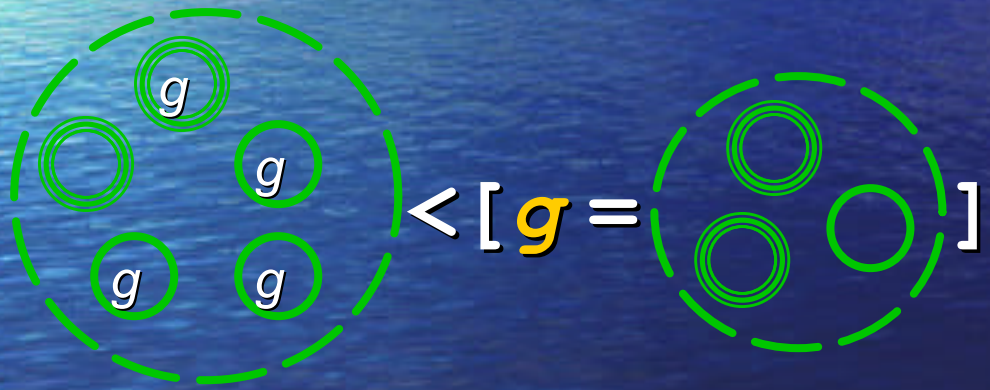
Monotone Transfer Function

- Plug: $x_1 \langle [g = x_2]$



Monotone Transfer Function

- Plug: $x_1 <[g = x_2]$



Monotone Transfer Function

- Plug: $x_1 <[g = x_2]$



Monotone Transfer Function

- Plug: $x_1 <[g = x_2]$



Monotone Transfer Function

- Plug: $x_1 <[g = x_2]$



However, too imprecise...!

Monotone Transfer Function

- Plug: $x_1 <[g = x_2]$



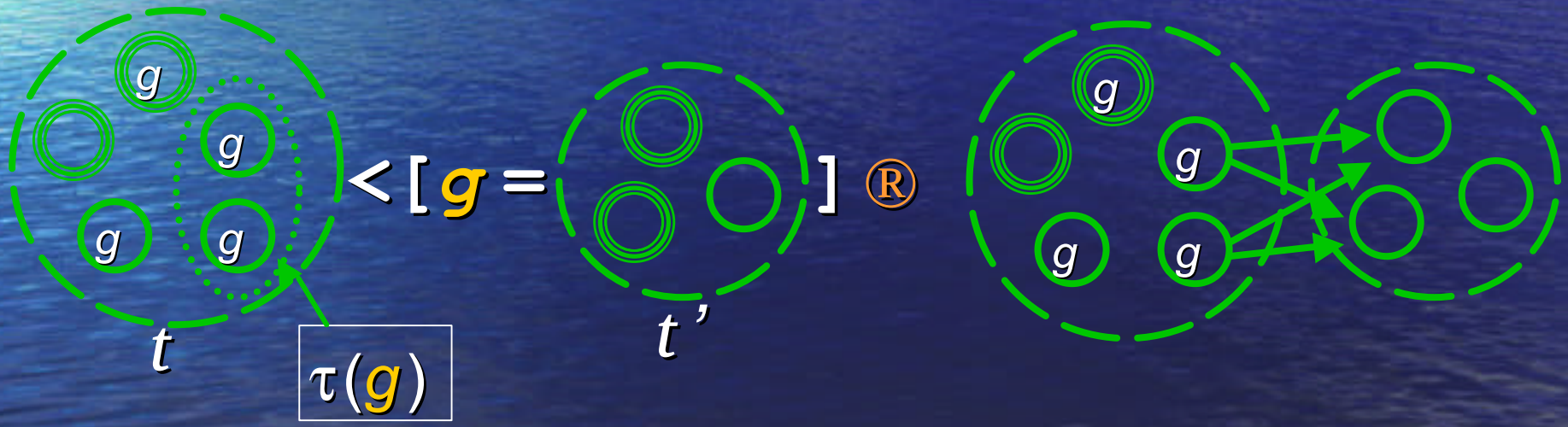
Monotone Transfer Function

- Plug: $x_1 <[g = x_2]$



GapTrack Analysis

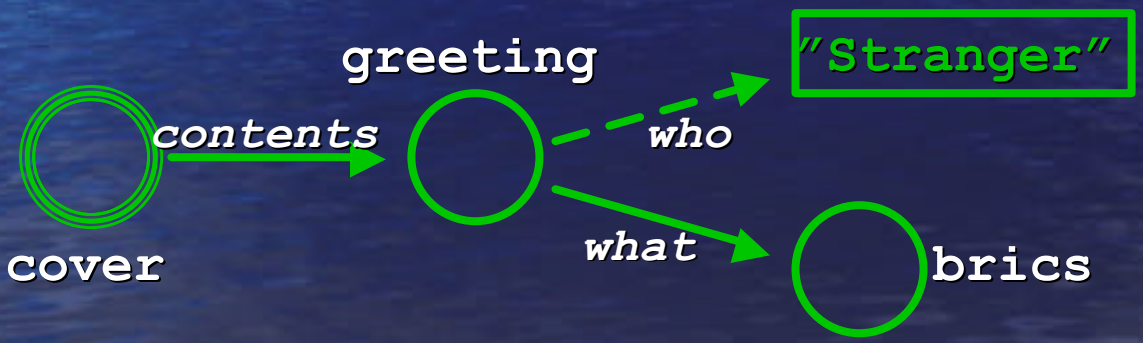
- GapTrack data-flow analysis:
  - ∀ program points compute: $\tau : GName \rightarrow 2^{TEMPLATES}$
    - "Tracks the origins (templates) of open gaps"



Validation?

- Now: summary graph ∀show statements
  - Then what...?

```
...;
h = greeting <[who = "Stranger"];
h = h <[what = <html><b>BRICS</b></html>];
show cover <[contents = h];
```



Validation?

A summary graph describes:



Validation?

A summary graph describes:

a set of *hopeful* XML docs

A DTD describes (e.g. XHTML 1.0):

a set of *valid* XML docs

Validation!

A summary graph describes:

a set of *hopeful* XML docs

$\subseteq$

validation!

A DTD describes (e.g. XHTML 1.0):

a set of *valid* XML docs

Validation!

- ***Decidable!***:
  - Summary graph traversal (parsing vs. DTD)
  - Memoization $\Rightarrow$ termination
  - Sound **and complete**!



Compiler Validation Recipe

- Step-by-Step:
  1. Extract control-flow graph
  2. **Gap-and-Field** data-flow analysis
  3. **GapTrack** data-flow analysis
  4. **Summary Graph** data-flow analysis
  5. **Validation** (∀shows) graph analysis

Experiments

800MHz Pentium III / Linux

Program	# Lines	# Templates	Time (sec.)
`chat`	65	3	0.1
`guess`	75	6	0.1
`calendar`	77	5	0.1
`xbiff`	561	18	0.1
`webboard`	1132	37	0.6
`cdshop`	1709	36	0.5
`jaoo`	1941	73	2.4
`bachelor`	2535	137	8.2
`courses`	4465	57	1.3
`eatcs`	5345	133	6.7

Many validation errors found, *no spurious errors*

Example Revisited

```
html greeting = <html>
  Hello <[who]>, welcome to <[what]>.
</html>;

html cover = <html>
  <head><title>Welcome</title></head>
  <body><[contents]></body>
</html>;

html h;
h = greeting <[who = "Stranger"];
h = h <[what = <html><b>BRICS</b></html>];
show cover <[contents = h];
```

Planting an Error

```
html greeting = <html>
  <td>Hello <[who]>, welcome to <[what]>.</td>
</html>;

html cover = <html>
  <head><title>Welcome</title></head>
  <body><table><[contents]></table></body>
</html>;

html h;
h = greeting <[who = "Stranger"];
h = h <[what = <html><b>BRICS</b></html>];
show cover <[contents = h];
```

Ex

```
--- welcome.wig:13 HTML Validation:
welcome.wig:7
  warning: possible illegal subelement 'td' of 'table'
  template: <table><[contents]></table>
  contents: td
  plugs: contents:{welcome.wig:13}
```

```
1  html greeting = <html>
2    <td>Hello <[who]>, welcome to <[what]>.</td>
3  </html>;
4
5  html cover = <html>
6    <head><title>Welcome</title></head>
7    <body><table><[contents]></table></body>
8  </html>;
9
10 html h;
11 h = greeting <[who = "Stranger"];
12 h = h <[what = <html><b>BRICS</b></html>];
13 show cover <[contents = h];
```

Problems?

- "DynDoc":
  - Templates with named gaps
  - Plug and show
- Problems:
  - ~~Forces linear document construction~~
  - ~~Intermixes programmer / designer aspects~~
  - ~~No interaction correspondence~~
  - No static XML validation
  - No common fragments caching

Problems?

- "DynDoc":
  - Templates with named gaps
  - Plug and show
- Problems:
  - ~~Forces linear document construction~~
  - ~~Intermixes programmer / designer aspects~~
  - ~~No interaction correspondence~~
  - ~~No static XML validation~~
  - No common fragments caching

Problems?

- "DynDoc":
  - Templates with named gaps
  - Plug and show
- Problems:
  - ~~Forces linear document construction~~
  - ~~Intermixes programmer / designer aspects~~
  - ~~No interaction correspondence~~
  - ~~No static XML validation~~
  - No common fragments caching

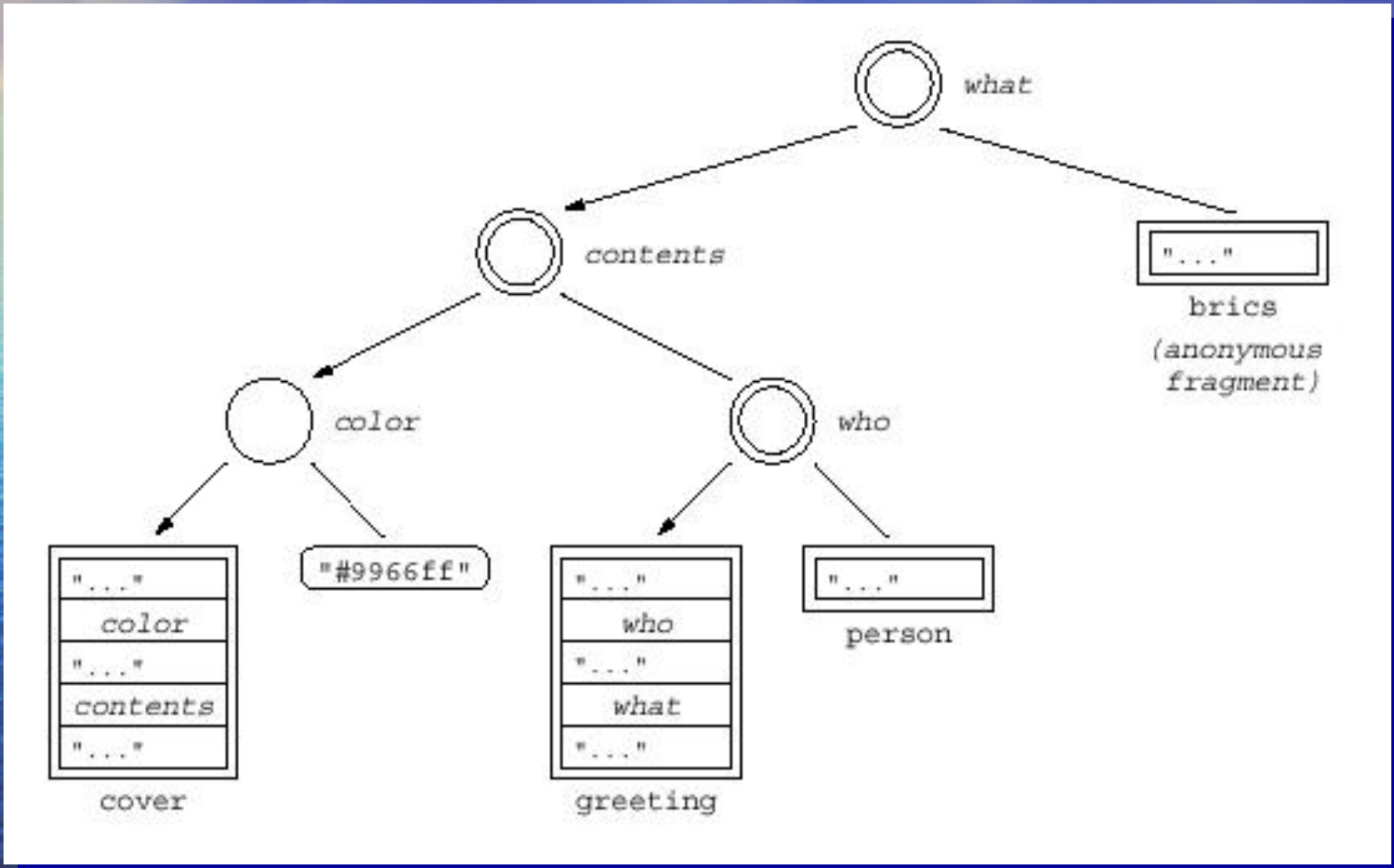
Outline

- Introduction
- Dynamic Documents
- Type Safety
- HTML Validation
- Caching
- Form-Field Validation
- Conclusion

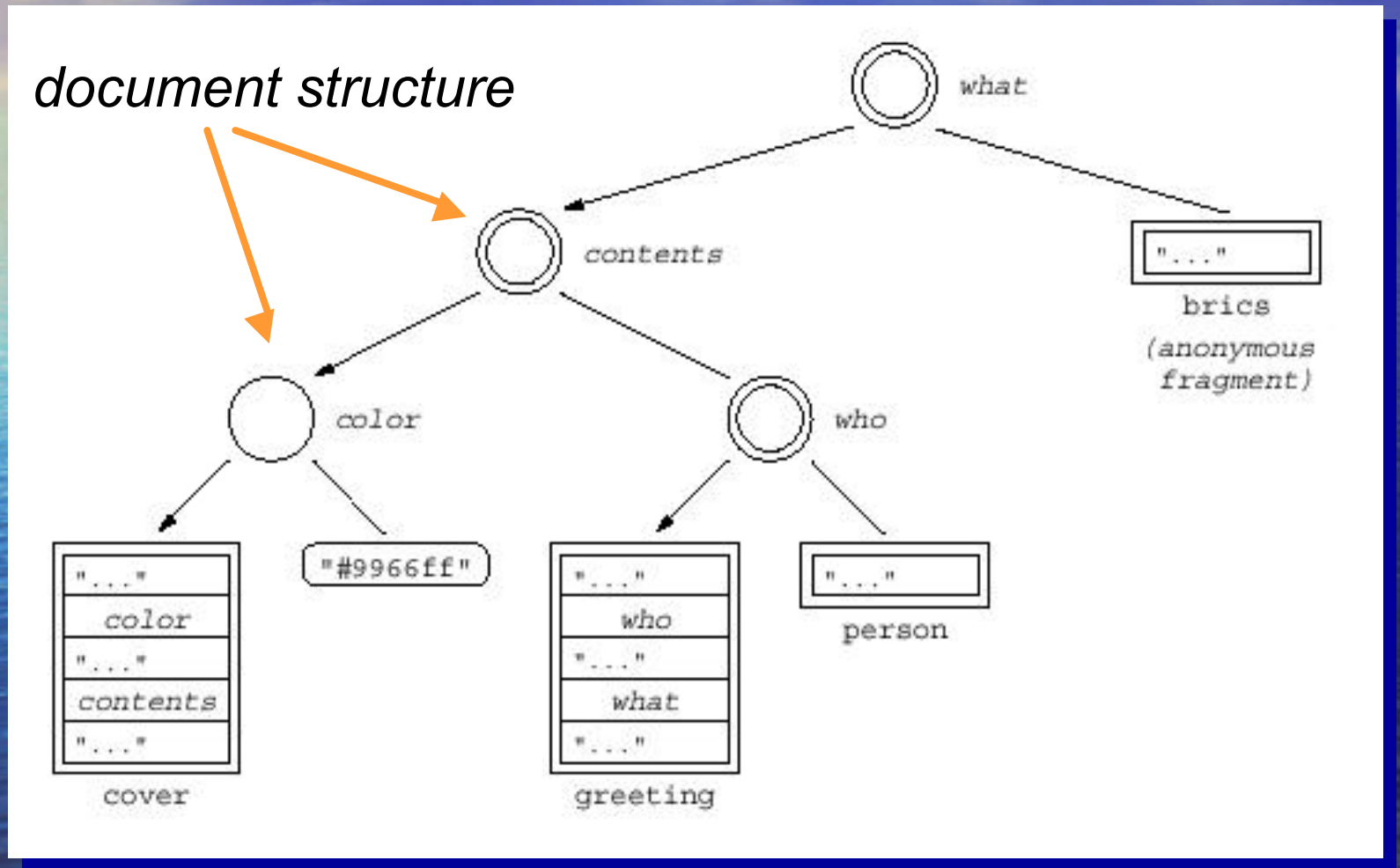
Outline

- Introduction
- Dynamic Documents
- Type Safety
- HTML Validation
- Caching
- Form-Field Validation
- Conclusion

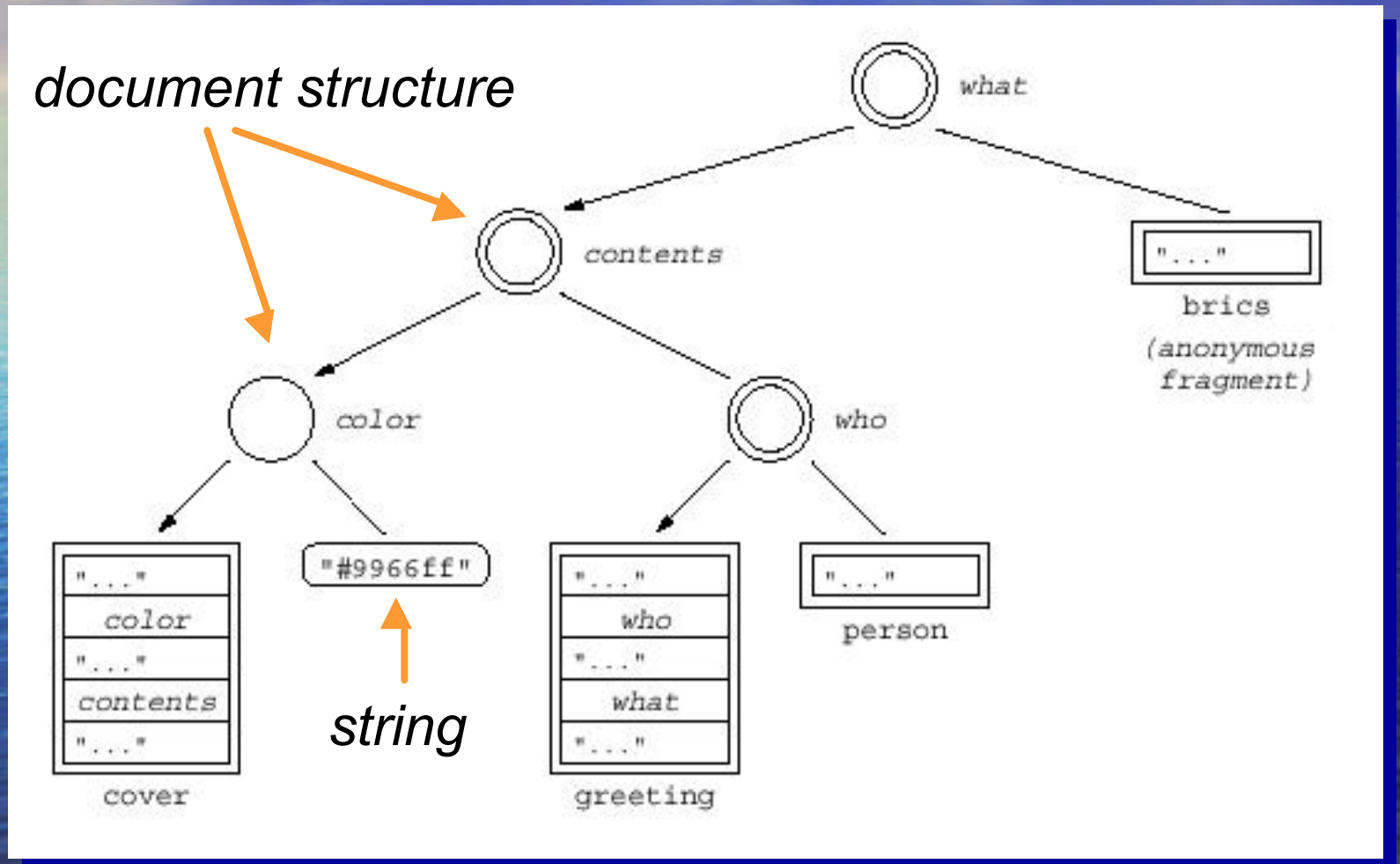
Document Representation



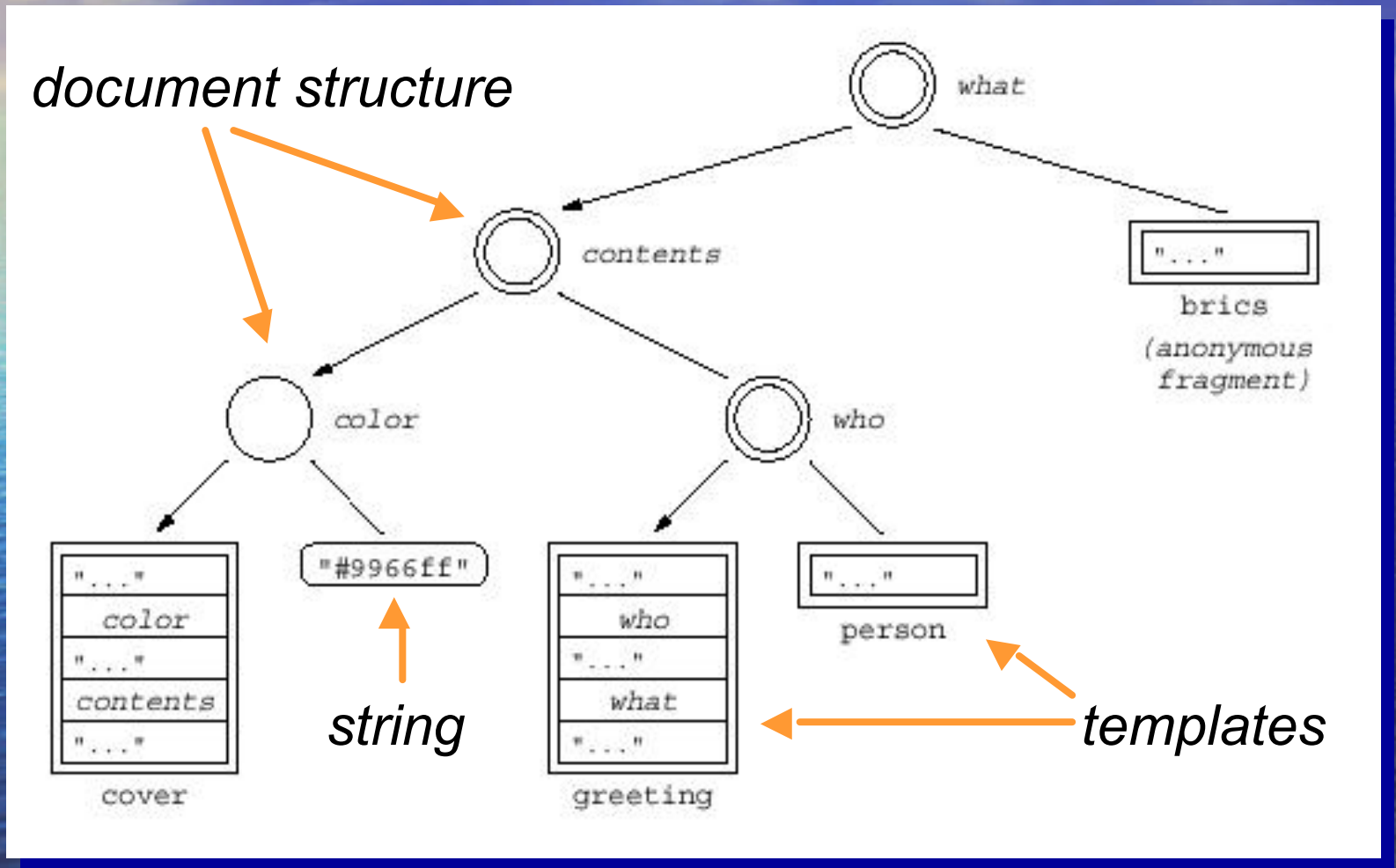
Document Representation



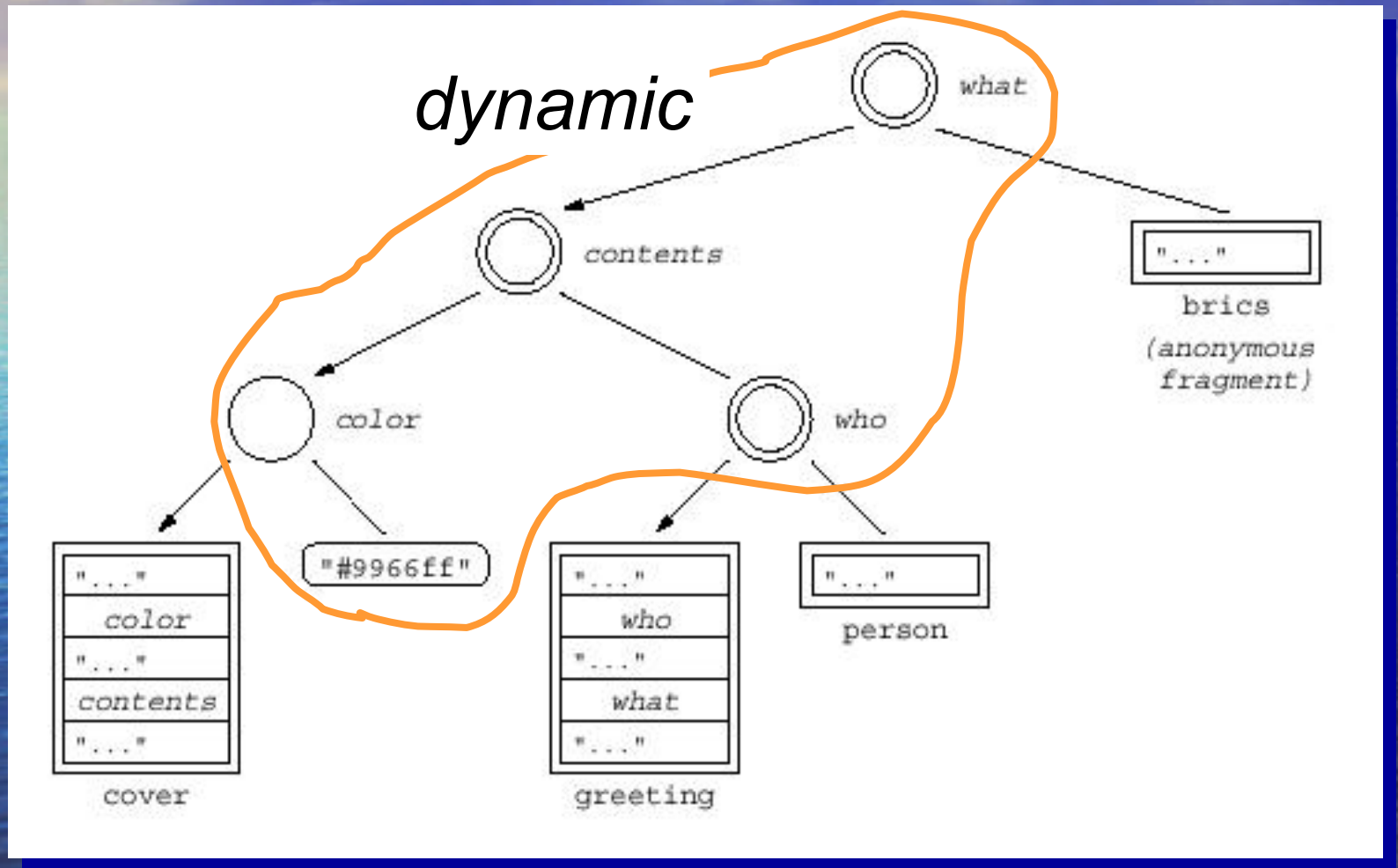
Document Representation



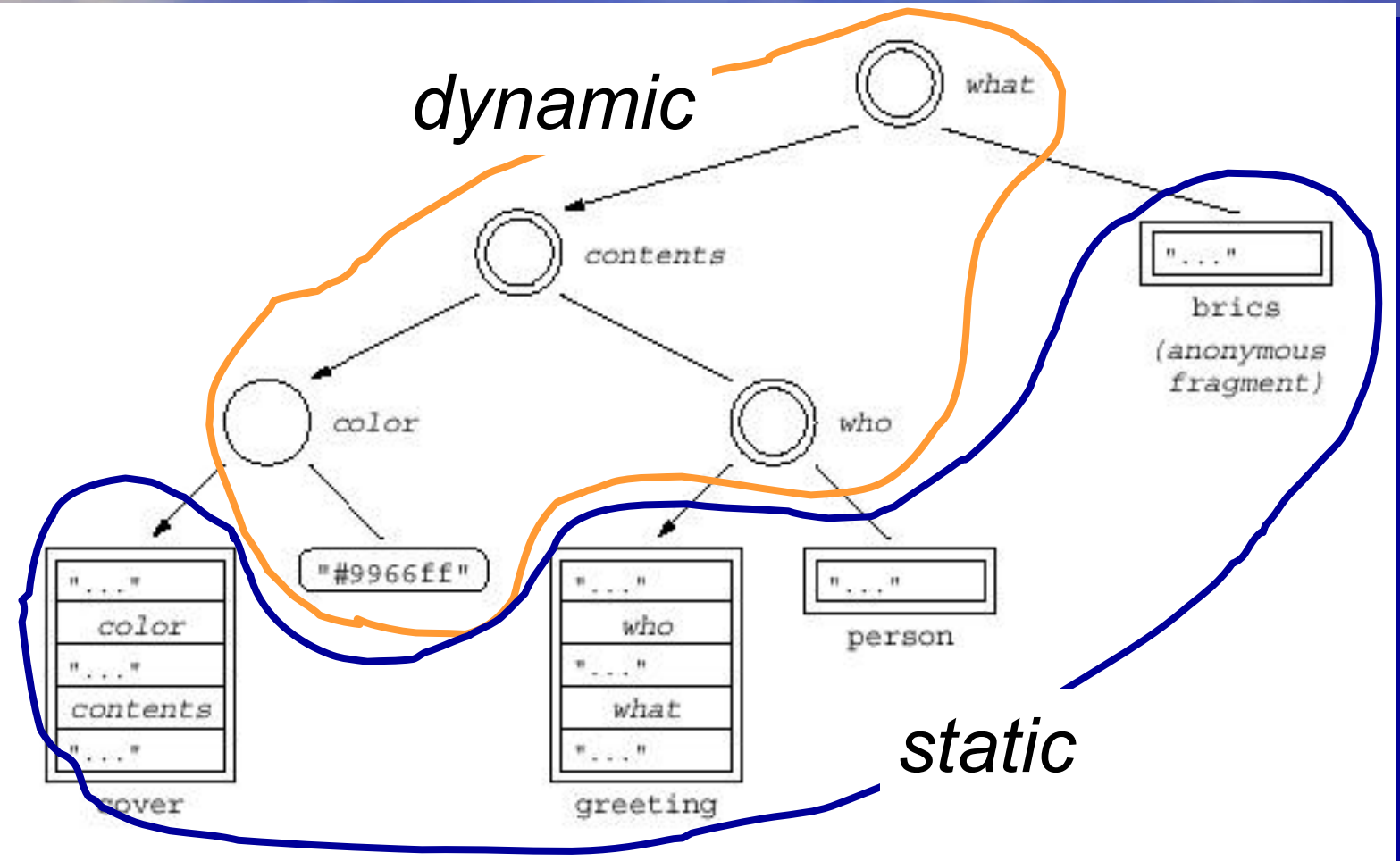
Document Representation



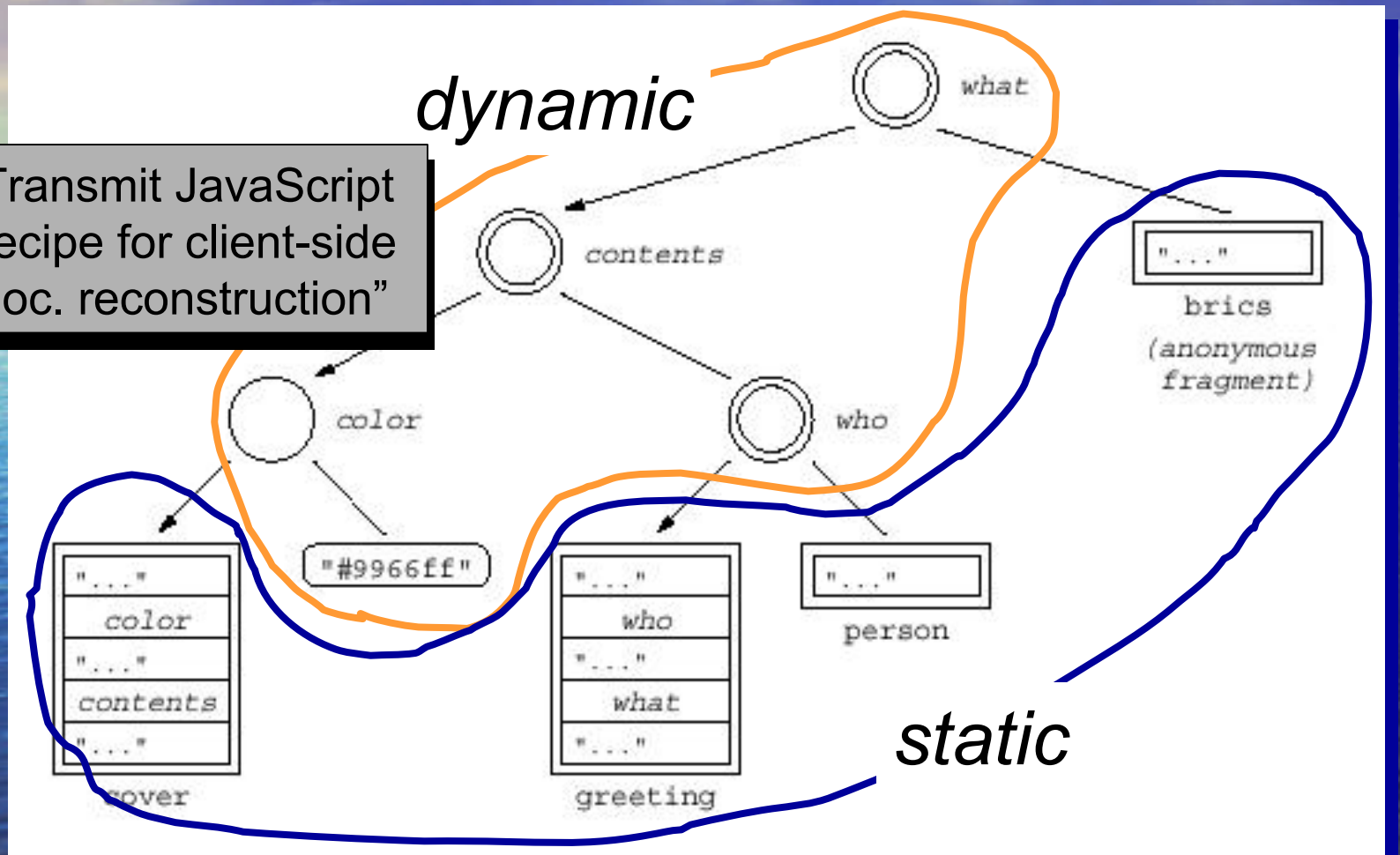
Document Representation



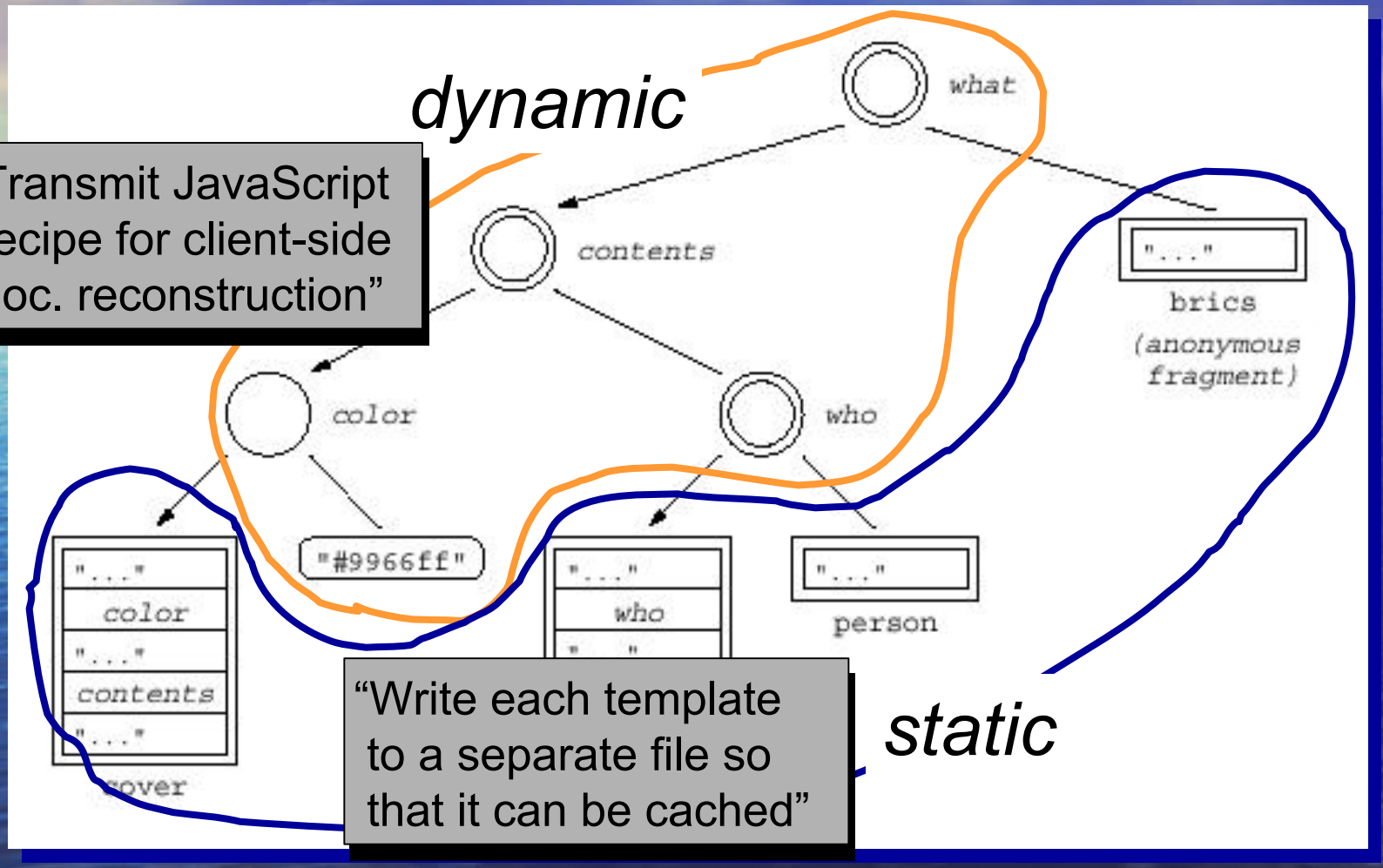
Document Representation



Document Representation

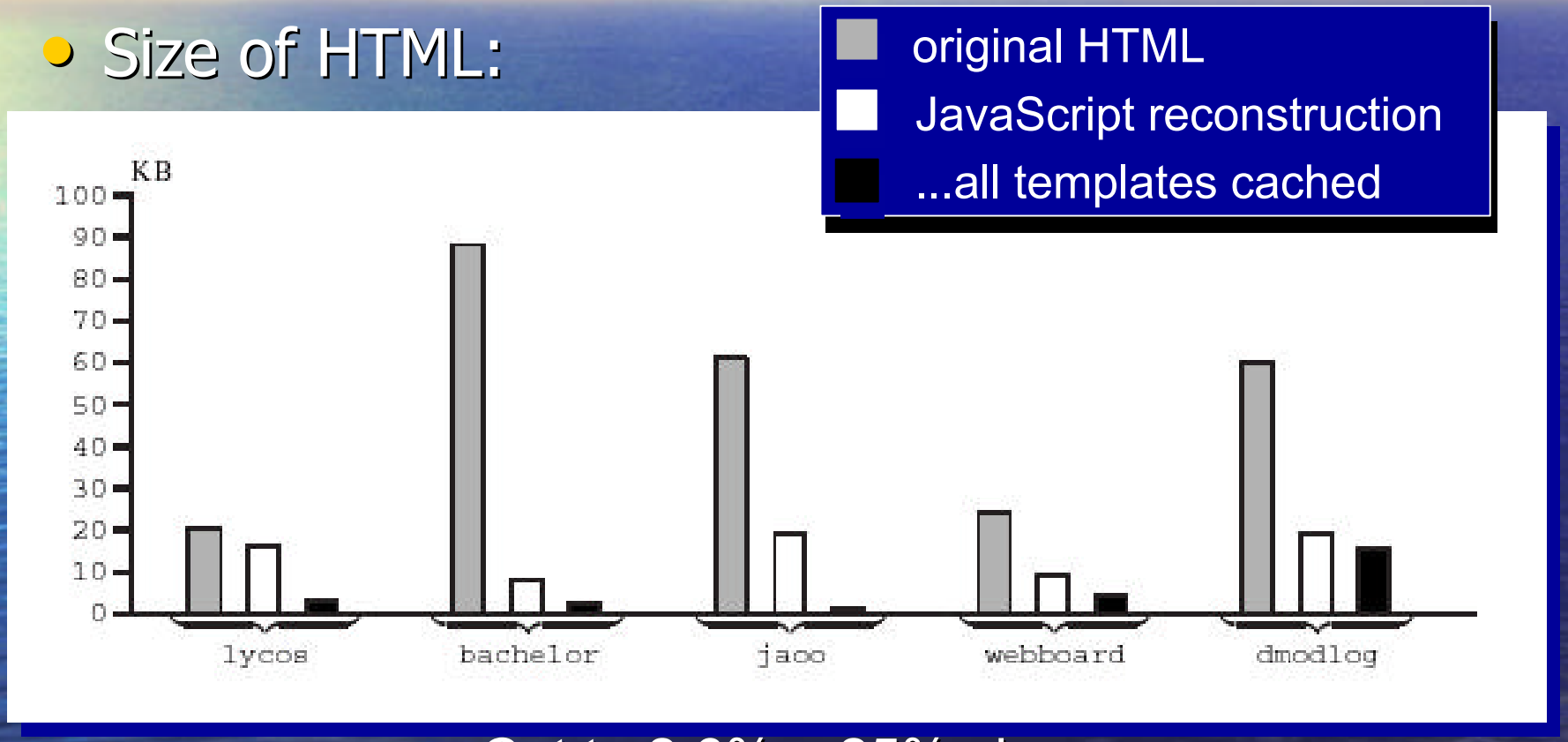


Document Representation



Experiments

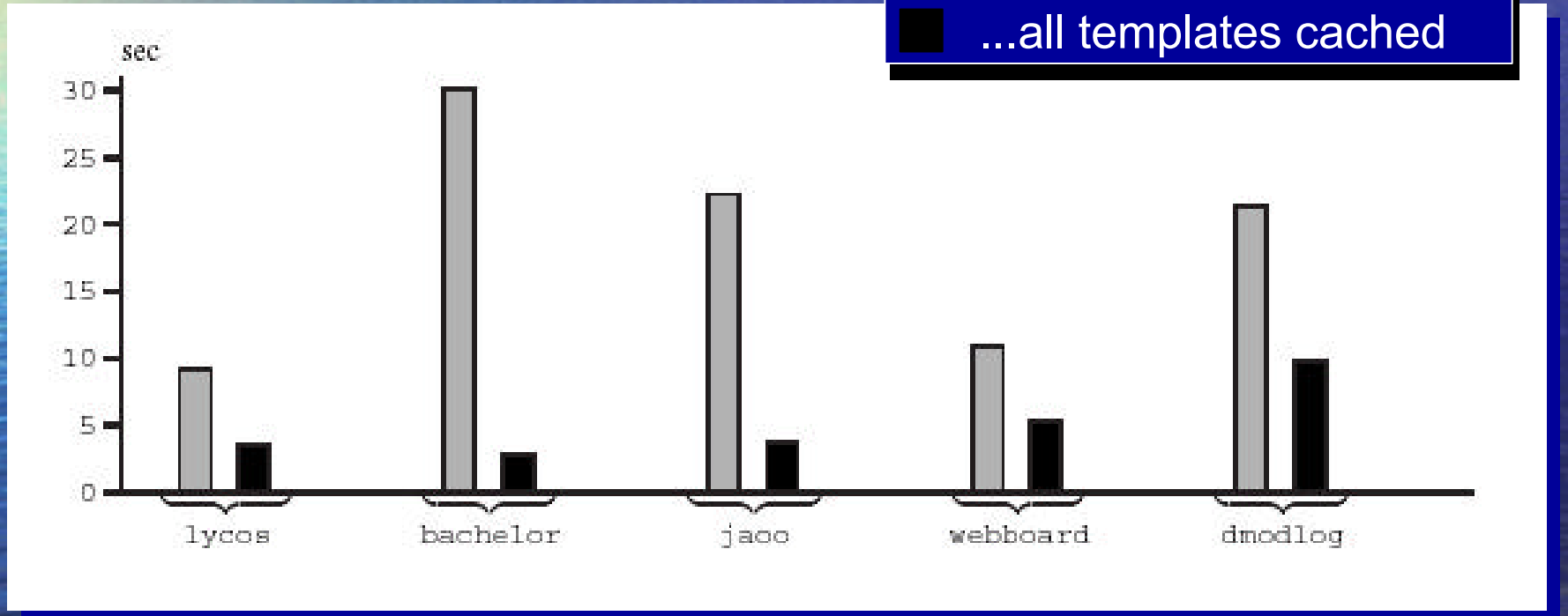
- Size of HTML:



Cut to 2.6% – 25% size

Experiments (700MHz, 28.8 Modem)

- Time (download+render):



2.2x – 10x faster

Problems?

- "DynDoc":
  - Templates with named gaps
  - Plug and show
- Problems:
  - ~~Forces linear document construction~~
  - ~~Intermixes programmer / designer aspects~~
  - ~~No interaction correspondence~~
  - ~~No static XML validation~~
  - No common fragments caching

Problems?

- "DynDoc":
  - Templates with named gaps
  - Plug and show
- Problems:
  - ~~Forces linear document construction~~
  - ~~Intermixes programmer / designer aspects~~
  - ~~No interaction correspondence~~
  - ~~No static XML validation~~
  - ~~No common fragments caching~~

Outline

- Introduction
- Dynamic Documents
- Type Safety
- HTML Validation
- Caching
- Form-Field Validation
- Conclusion

Outline

- Introduction
- Dynamic Documents
- Type Safety
- HTML Validation
- Caching
- Form-Field Validation
- Conclusion

Example: "EnterEmail"

```
html input = <html>
  Please enter your email:<br>
  <input type="text" name="email">
</html>;

html output = <html>Your email is: <[email]></html>;

string s;
show input receive[s = email];
show output <[email = s];
```

Example: "EnterEmail"

```
format Email = regexp("<word>@<word>(\.<word>)+");

html input = <html>
  Please enter your email:<br>
  <input type="text" name="email">
</html>;

html output = <html>Your email is: <[email]></html>;

string s;
show input receive[s = email];
show output <[email = s];
```

Example: "EnterEmail"

```
format Email = regexp("<word>@<word>(\.<word>)+");

html input = <html>
  Please enter your email:<br>
  <input type="text" name="email" format="Email">
</html>;

html output = <html>Your email is: <[email]></html>;

string s;
show input receive[s = email];
show output <[email = s];
```

Example: "EnterEmail"

```
format Email = regexp("<word>@<word>(\.<word>)+");

html input = <html>
  Please enter your email:<br>
  <input type="text" name="email" format="Email">
</html>;

html output = <html>Your email is: <[email]></html>;

string s;
show input receive[s = email];
show output <[email = s];
```

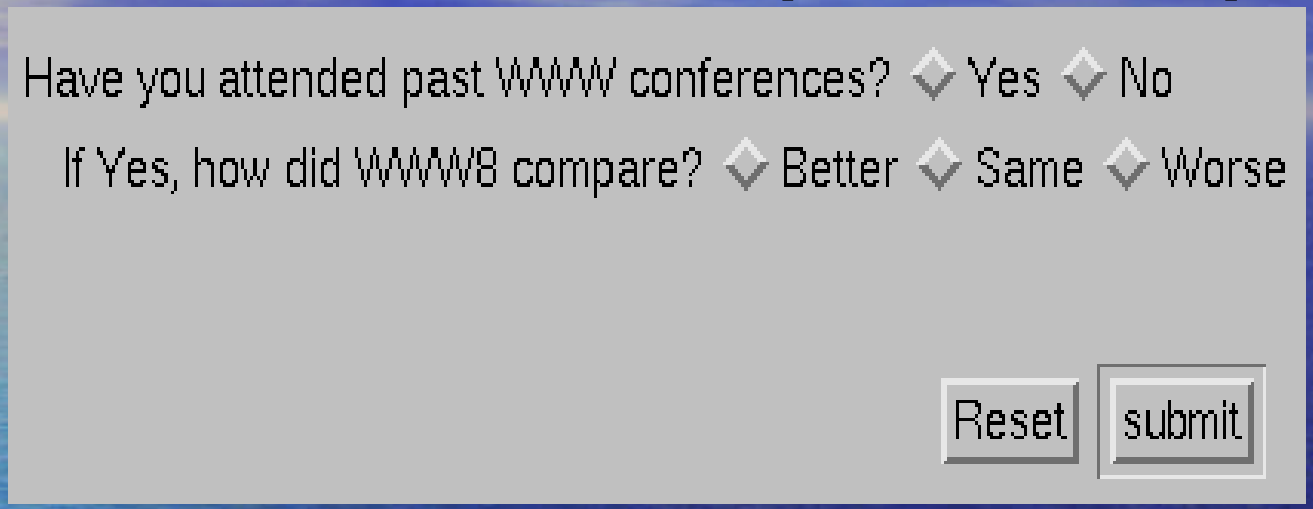
Form Field Validation: "PowerForms"

- Domain specific language:
  - uniquely for form-field validation
- Declarative specification (regexps):
  - Abstracts away operational details



- PowerForms also available as stand-alone tool

Form-Field Interdependency



- "Favorite Letter"
  - Select filtering
- "NYC Office"
  - Complex interdependency

Outline

- Introduction
- Dynamic Documents
- Type Safety
- HTML Validation
- Caching
- Form-Field Validation
- Conclusion

Outline

- Introduction
- Dynamic Documents
- Type Safety
- HTML Validation
- Caching
- Form-Field Validation
- Conclusion

<bigwig> Dynamic Documents + Static Analyses:

- Yield:

Flexible, Safe, and Efficient Dynamic Generation of XML

<bigwig> Dynamic Documents + Static Analyses:

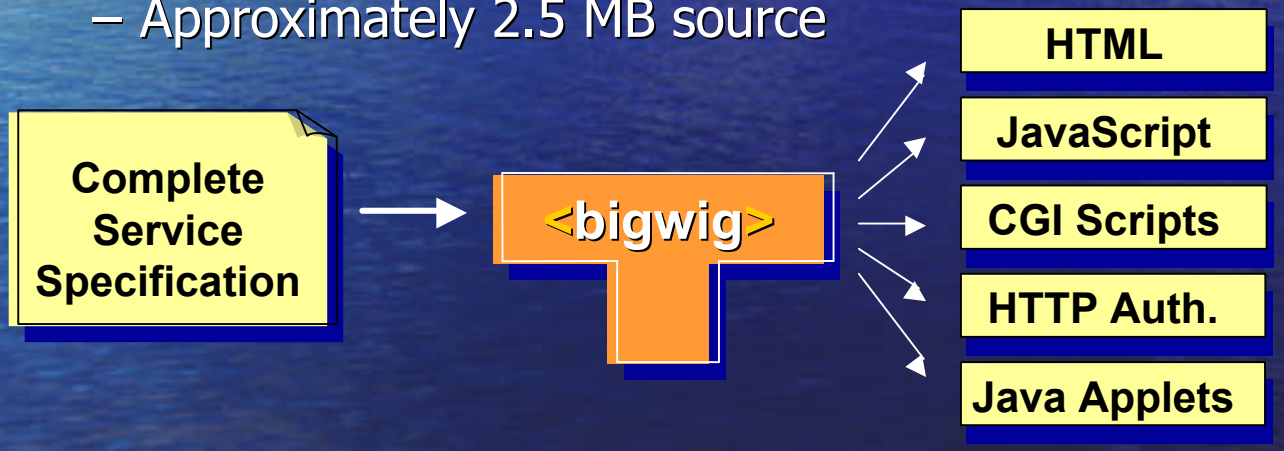
- Yield:

  Flexible, Safe, and Efficient Dynamic Generation of XML

- AND:
  - Non-linear document construction
  - Separates programmer / designer aspects
  - Guaranteed interaction correspondence
  - Static XML validation
  - Common fragments caching

The <bigwig> Compiler

- Version 2.0:
  - Implemented in C (for UNIX / Linux)
  - Complete source code available
    - Approximately 2.5 MB source



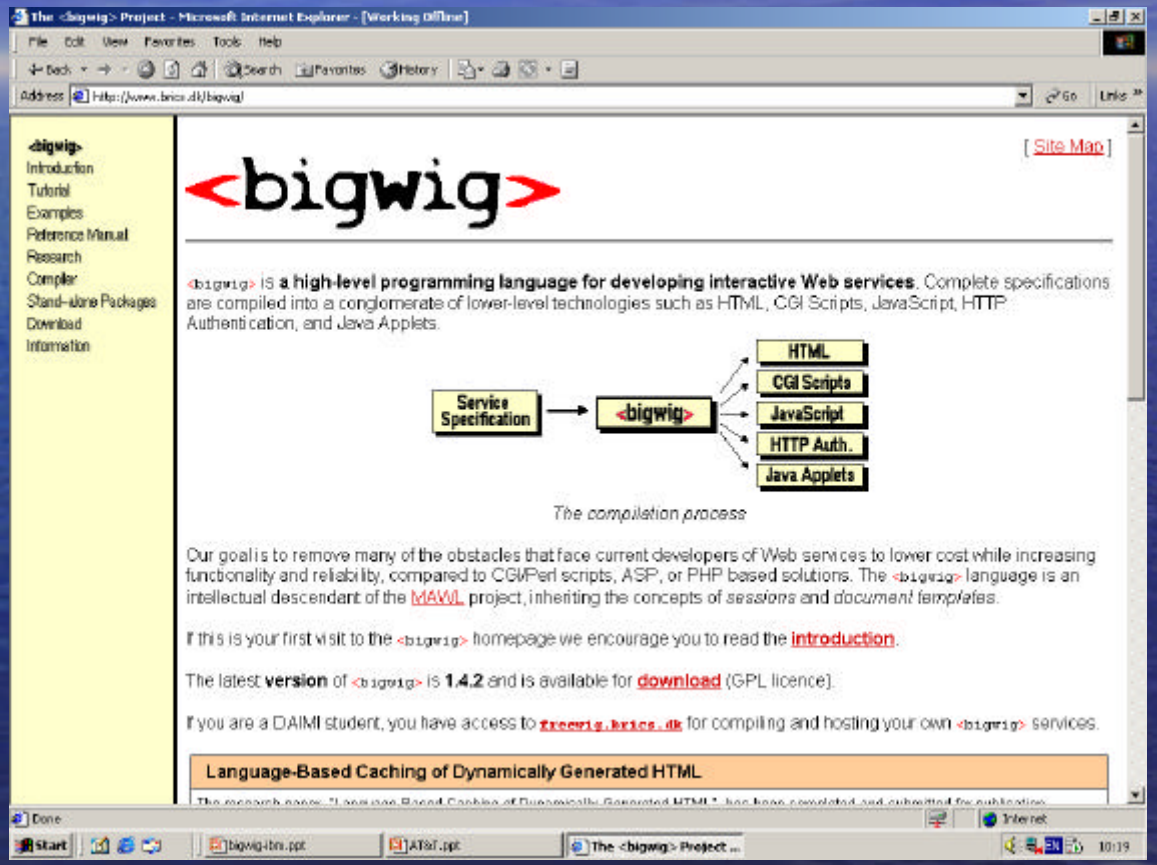
- License: GPL (Gnu Public License)

<bigwig> Publications

- Concurrency Control — FASE, 1998
- Runtime System — WWW 8, 1999
- Dynamic Documents — POPL, 2000
- Form-Field Validation — WWW Journal, 2000
- HTML Validation — PASTE, 2001
- The <bigwig> Project — TOIT, 2002
- Syntax Macros — PEPM, 2002
- Document Caching — WWW Journal, 2002
- ...

http://www.brics.dk/bigwig/

- Homepage:
  - Introduction
  - Examples
  - Ref. manual
  - Tutorials
  - Papers
  - Presentations
  - Downloads:
    - src / bin
  - 2 Recorded Presentations (MS & IBM Research)



<bigwig> → JWIG (Java)

http://www.brics.dk/JWIG/





Nova: A Programmable Platform for Multimedia Communication Services

Charles Consel

(joint work with members of Compose)

Compose Research Group

LaBRI /INRIA-Futurs

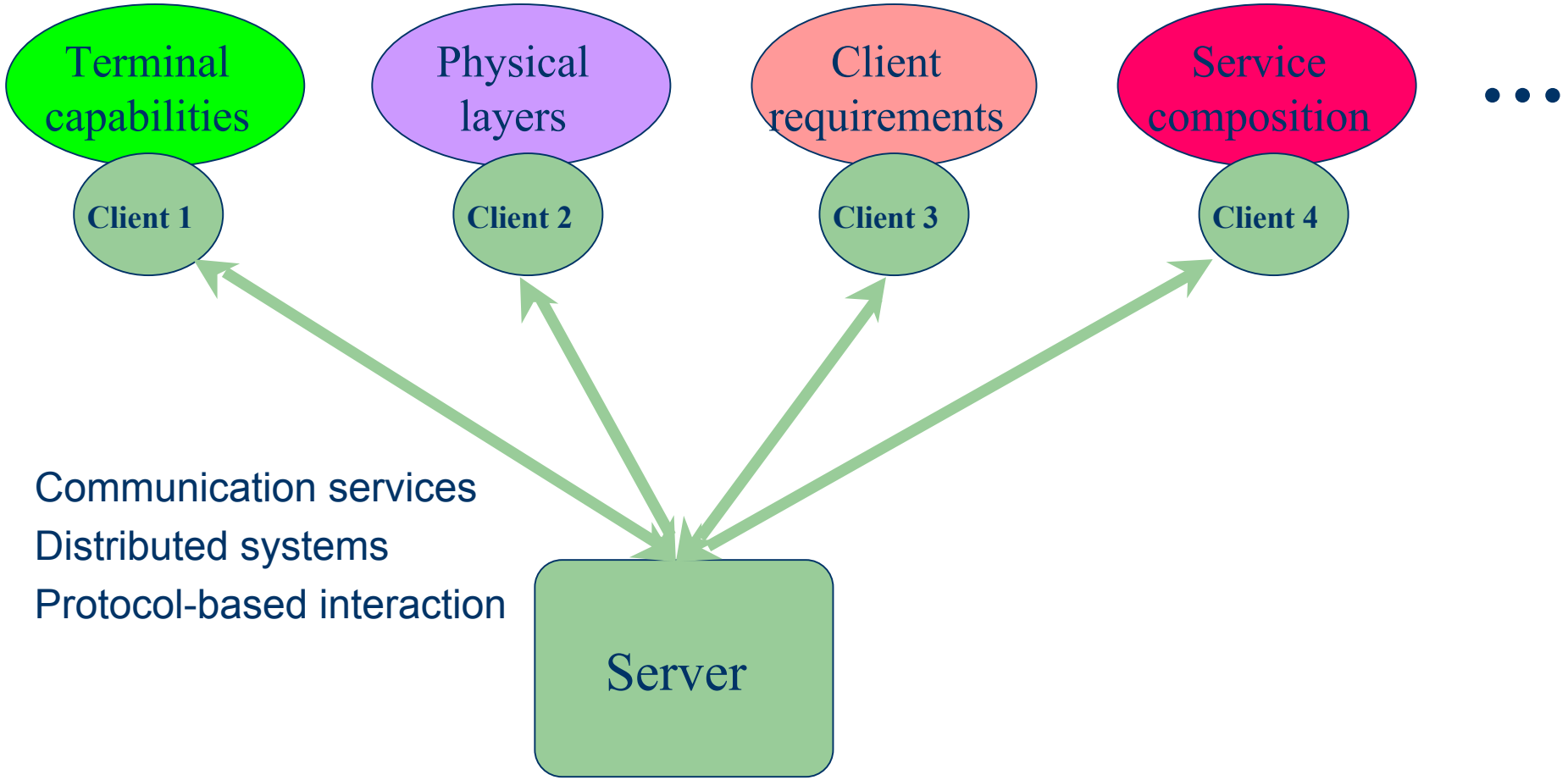
September 2003

Communication Services: A Rapidly Evolving Area



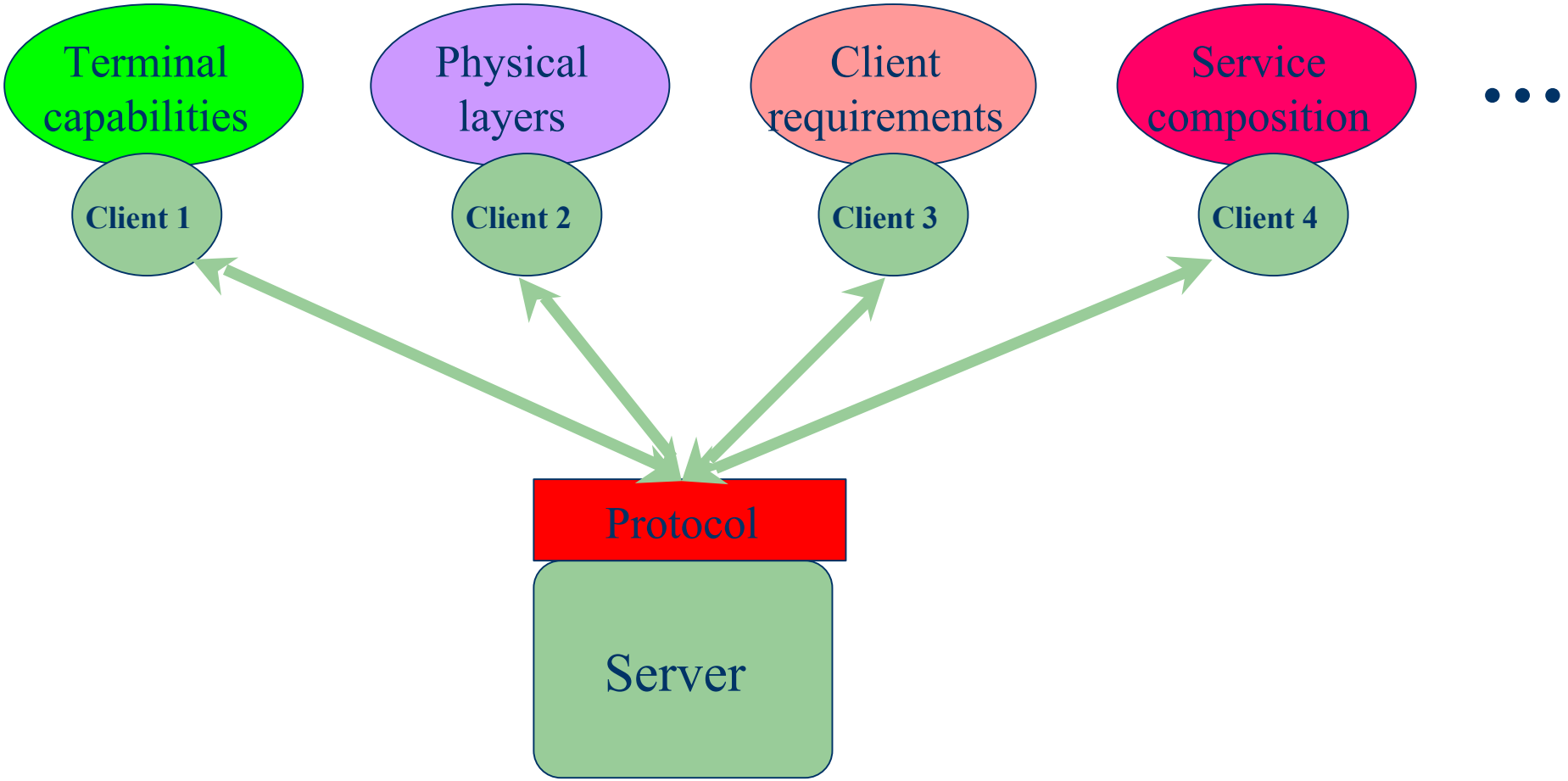
- E-mail
- HTTP
- Telephony
- Alert
- Chat
- …

Client-Server Model

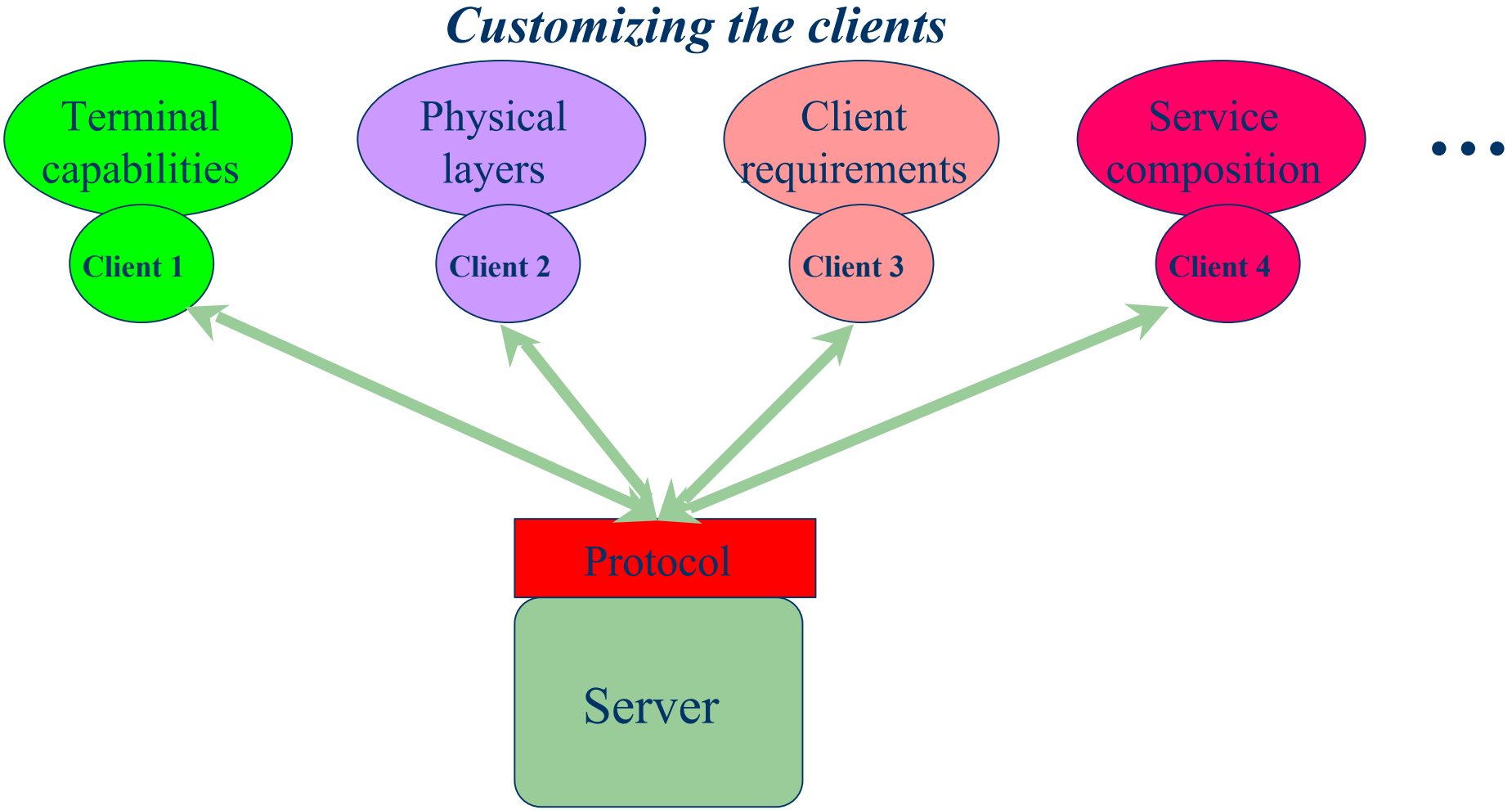


- Communication services
- Distributed systems
- Protocol-based interaction

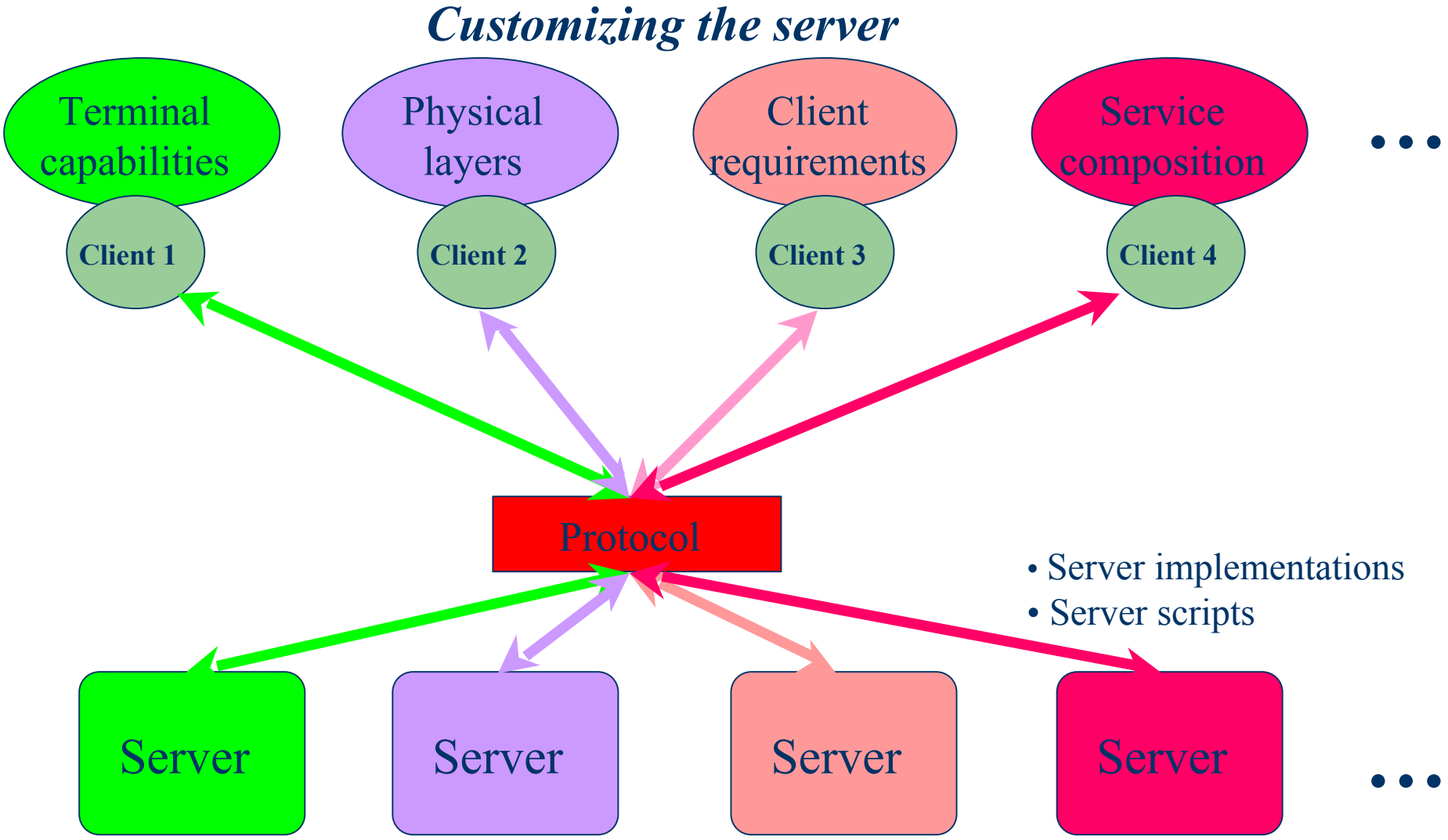
Client-Server Model: Limitations



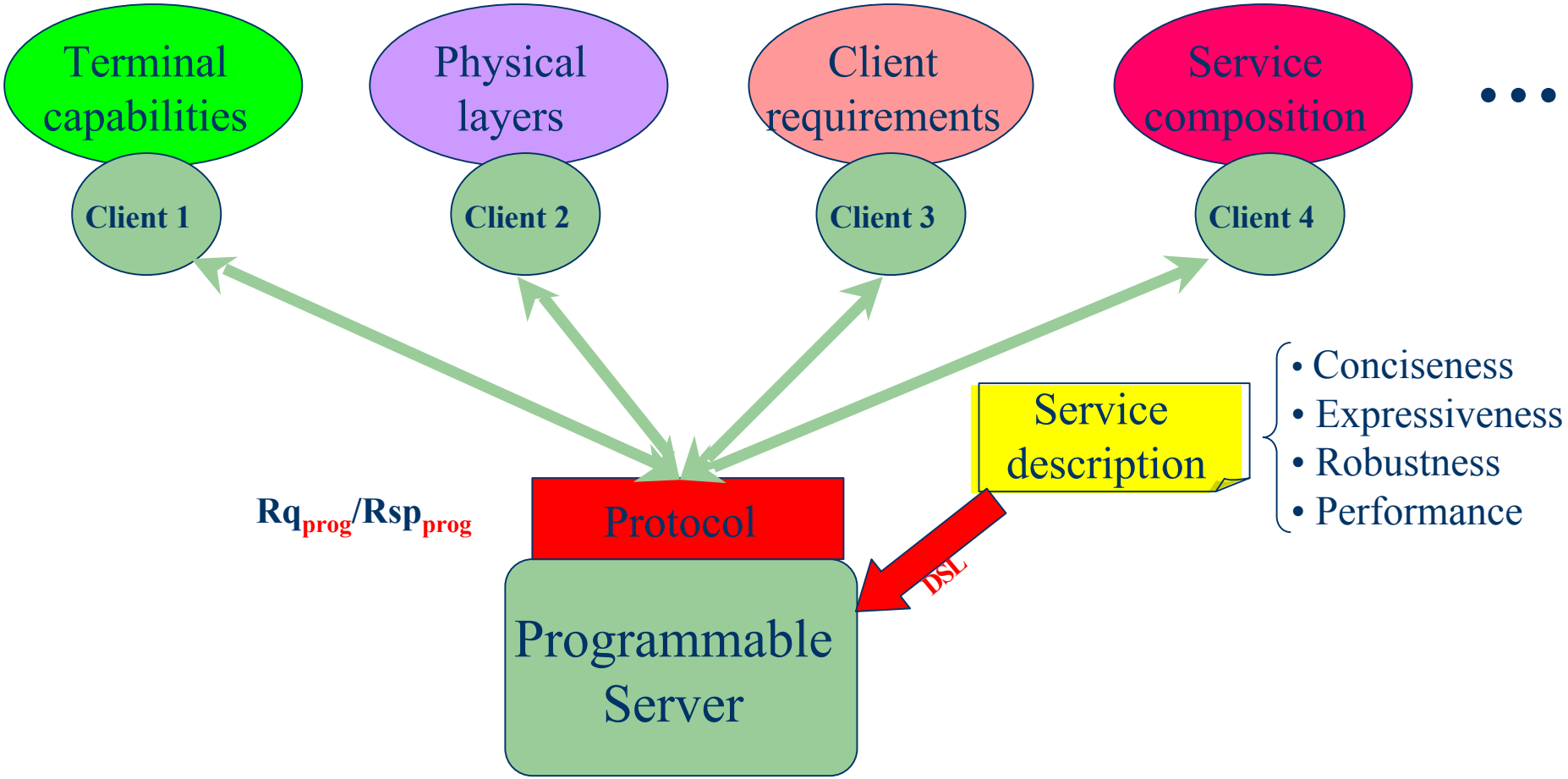
Overcoming the Limitations of the Client-Server Model



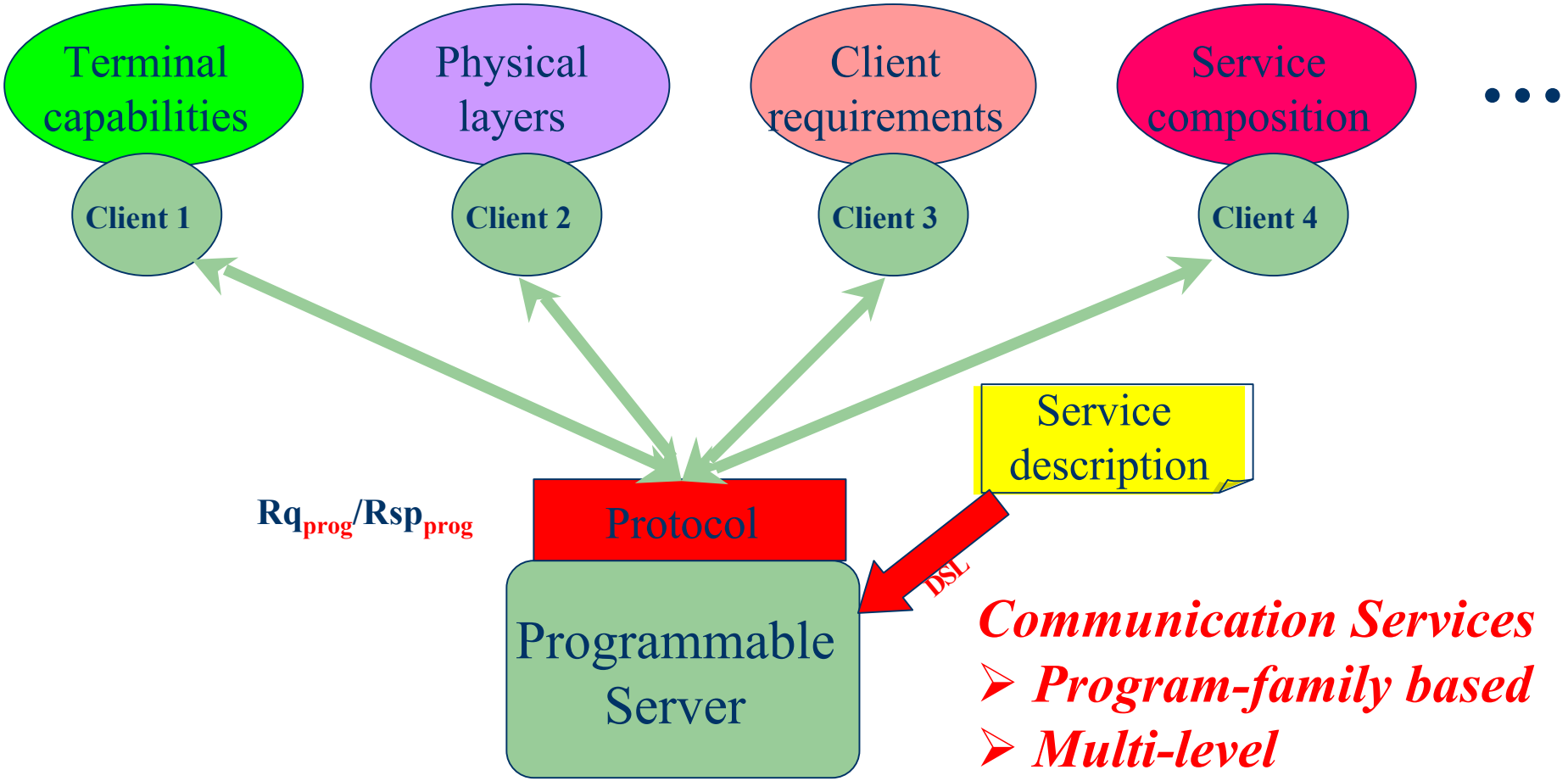
Overcoming the Limitations of the Client-Server Model



Making the Server Programmable



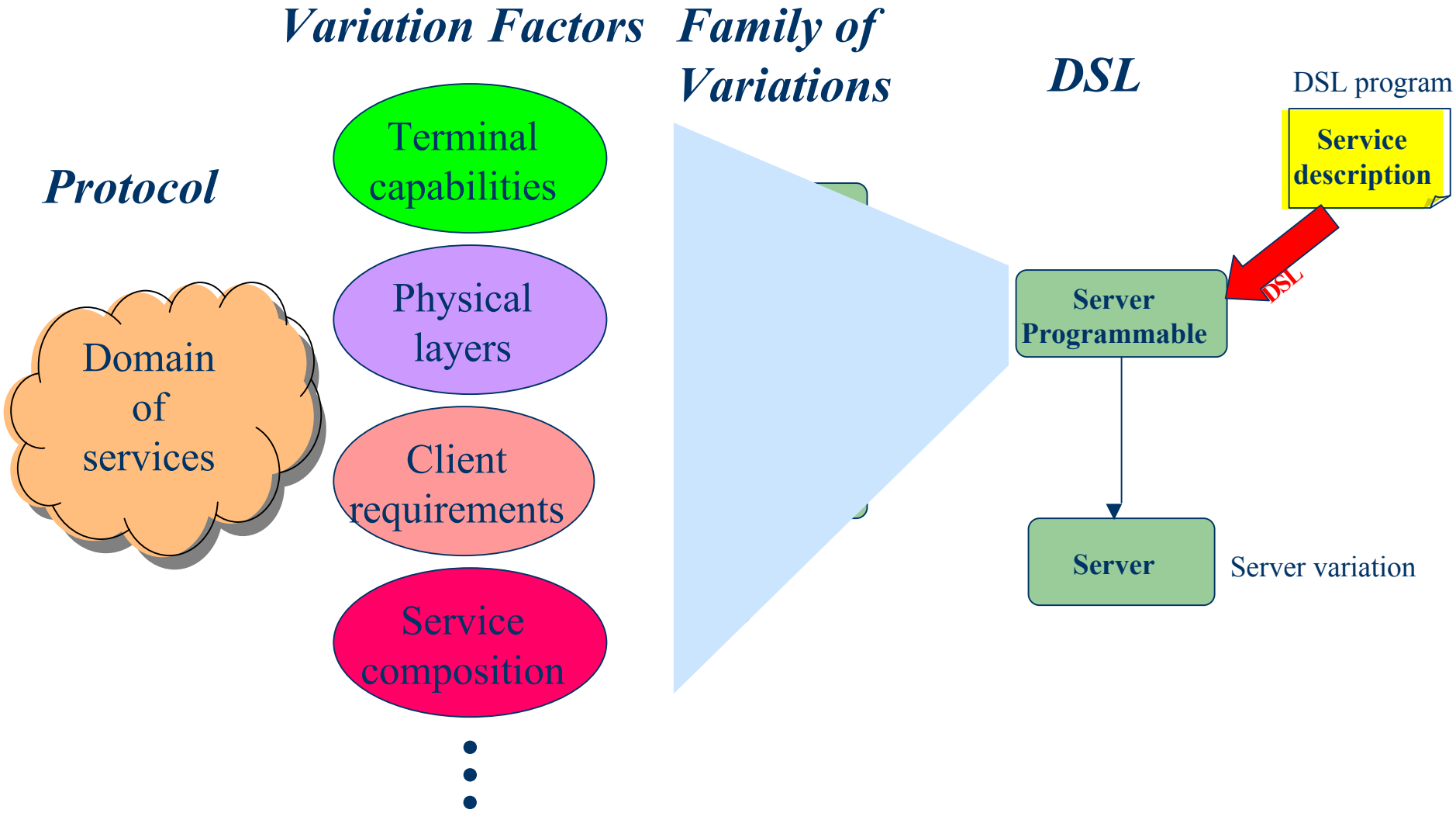
Making the Server Programmable



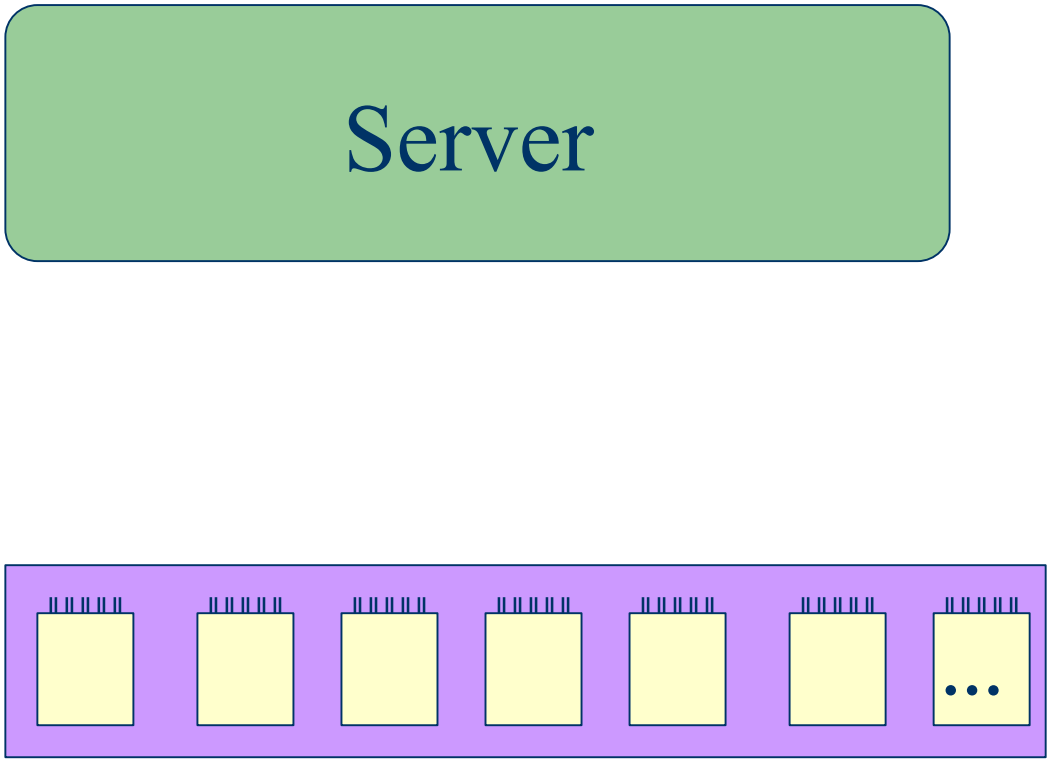
Communication Services

- ***Program-family based***
- ***Multi-level***
- ***Case studies***

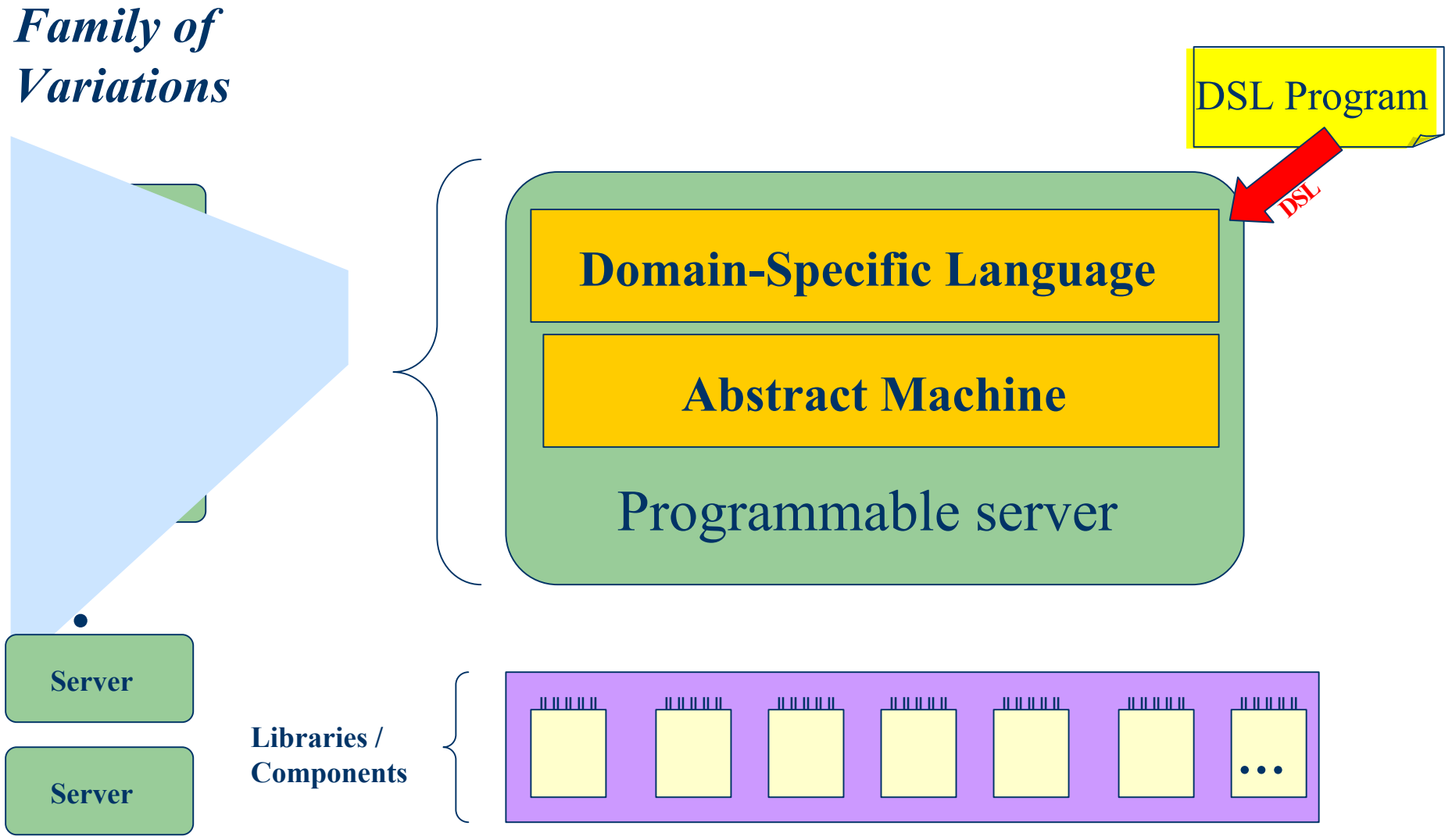
Program Family-Based Approach



Program Family-Based Approach

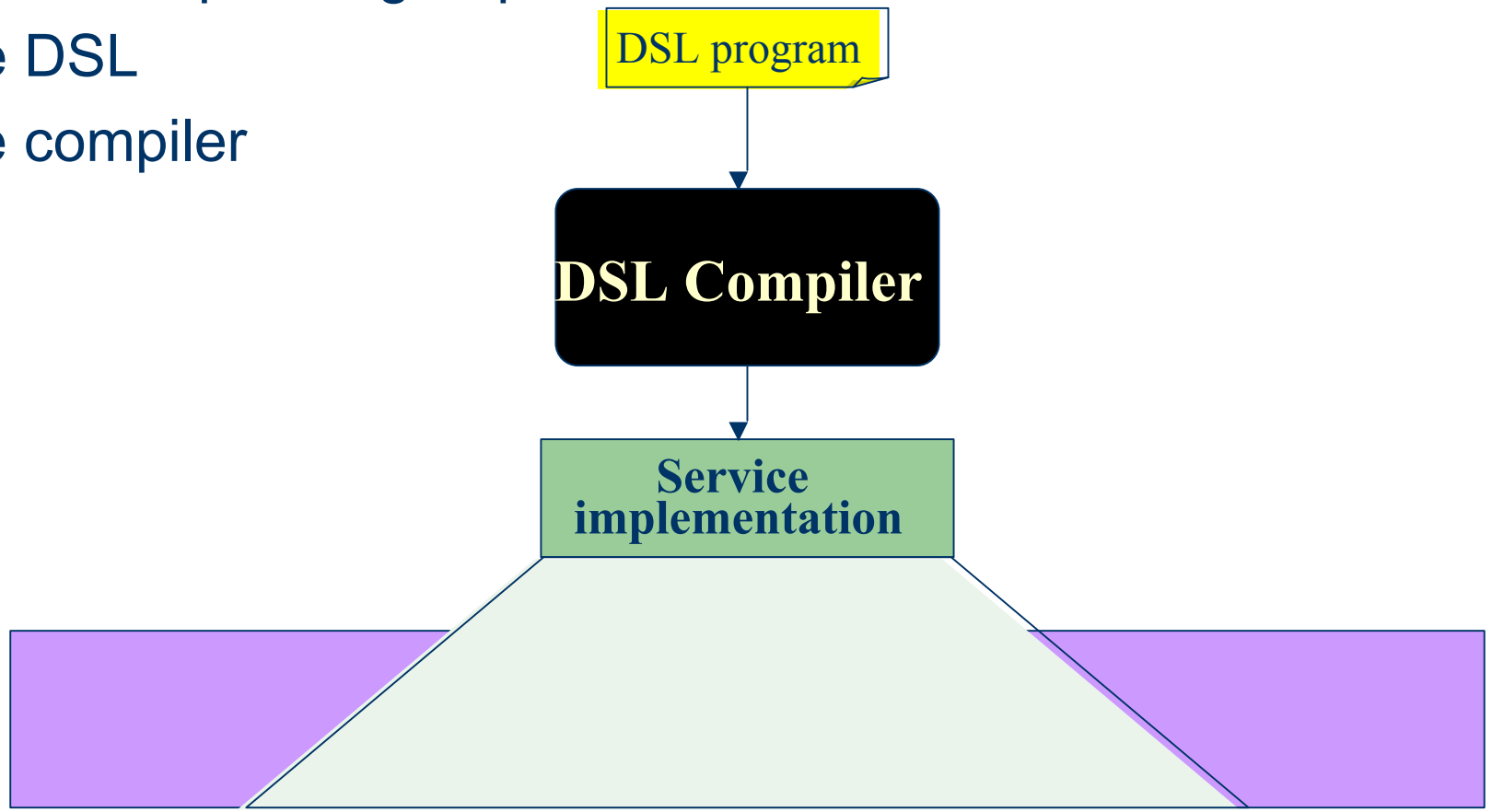


Program Family-Based Approach



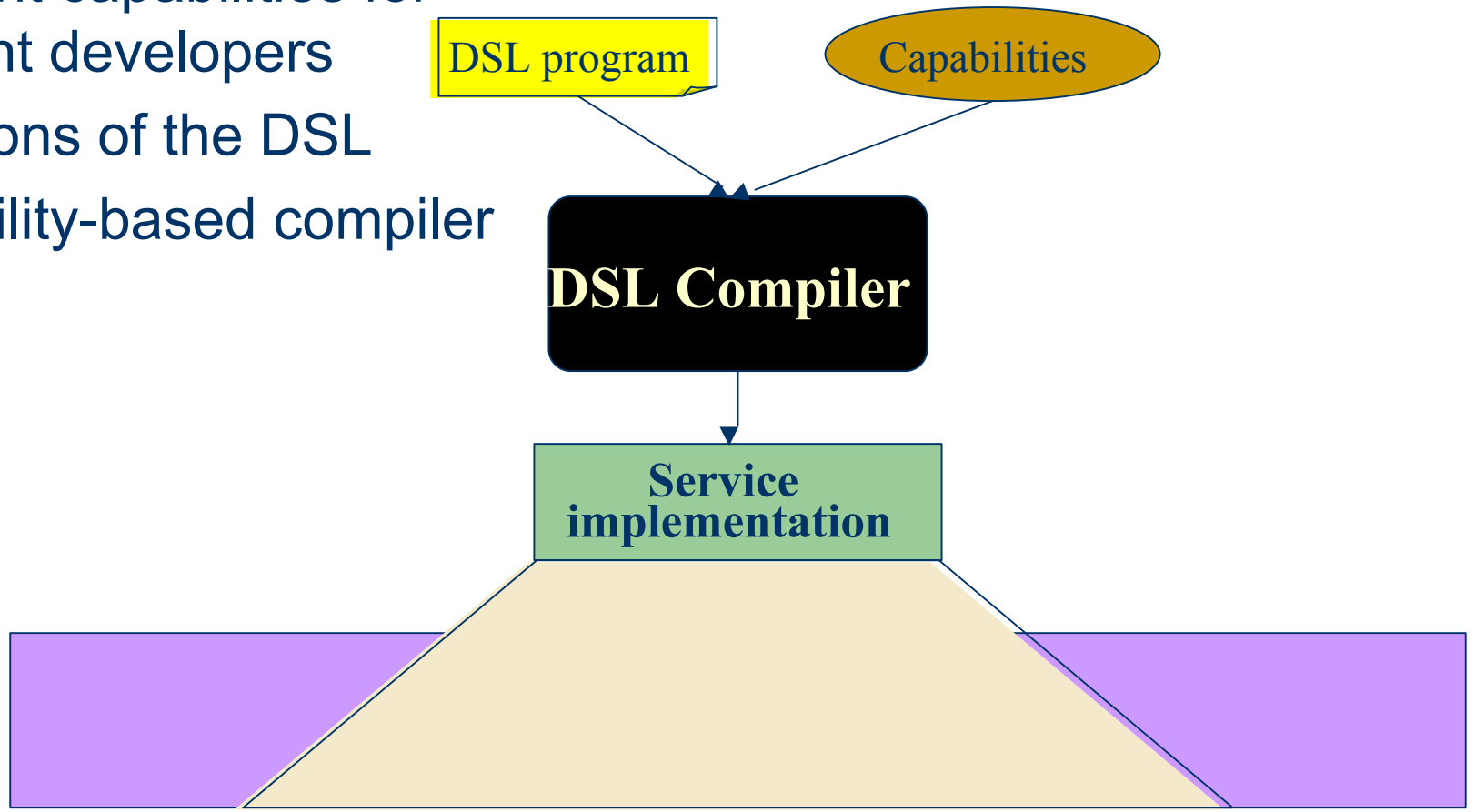
Mono-Level Programming

- Unique development group
- Unique DSL
- Unique compiler



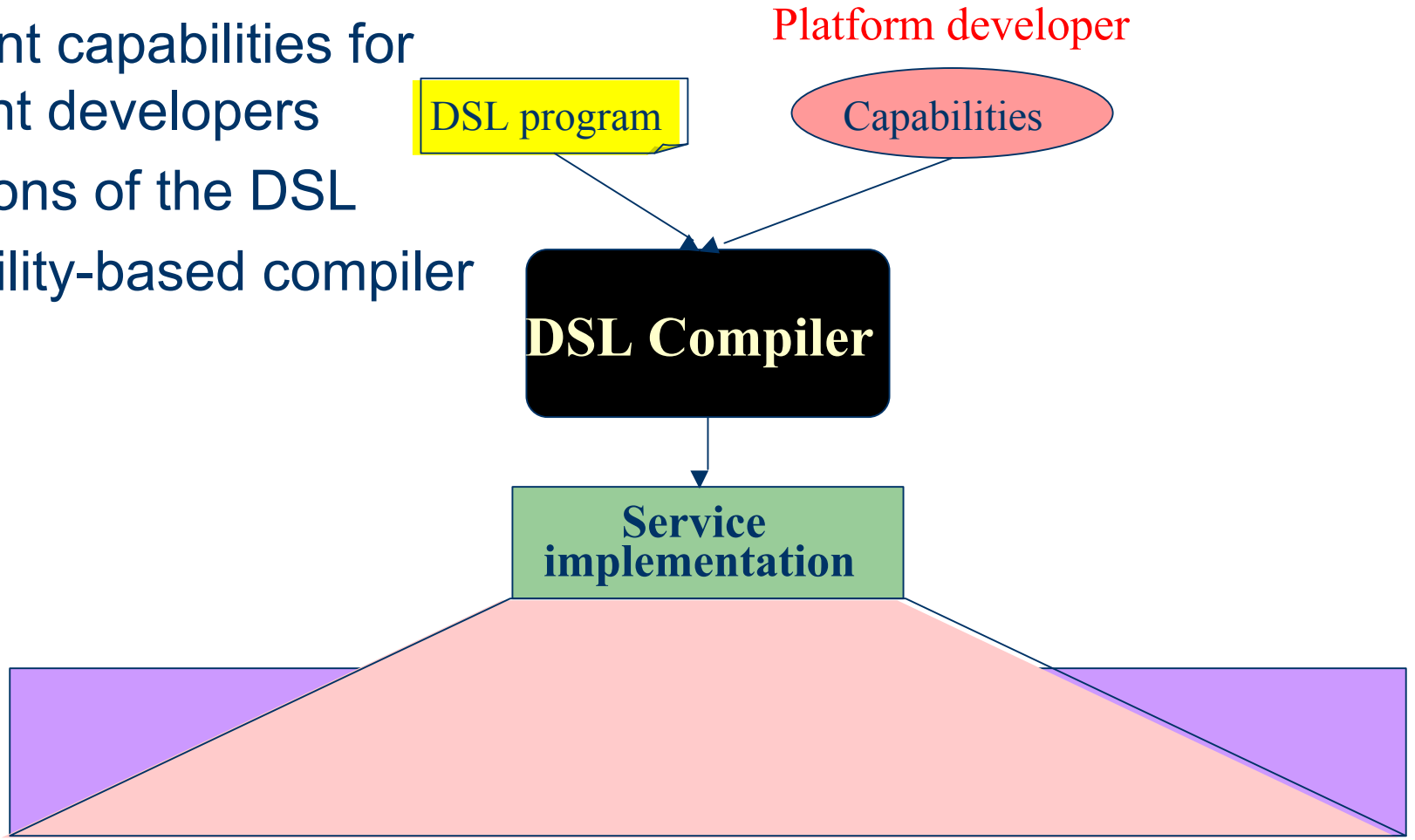
Multi-Level Programming

- Different capabilities for different developers
- Variations of the DSL
- Capability-based compiler



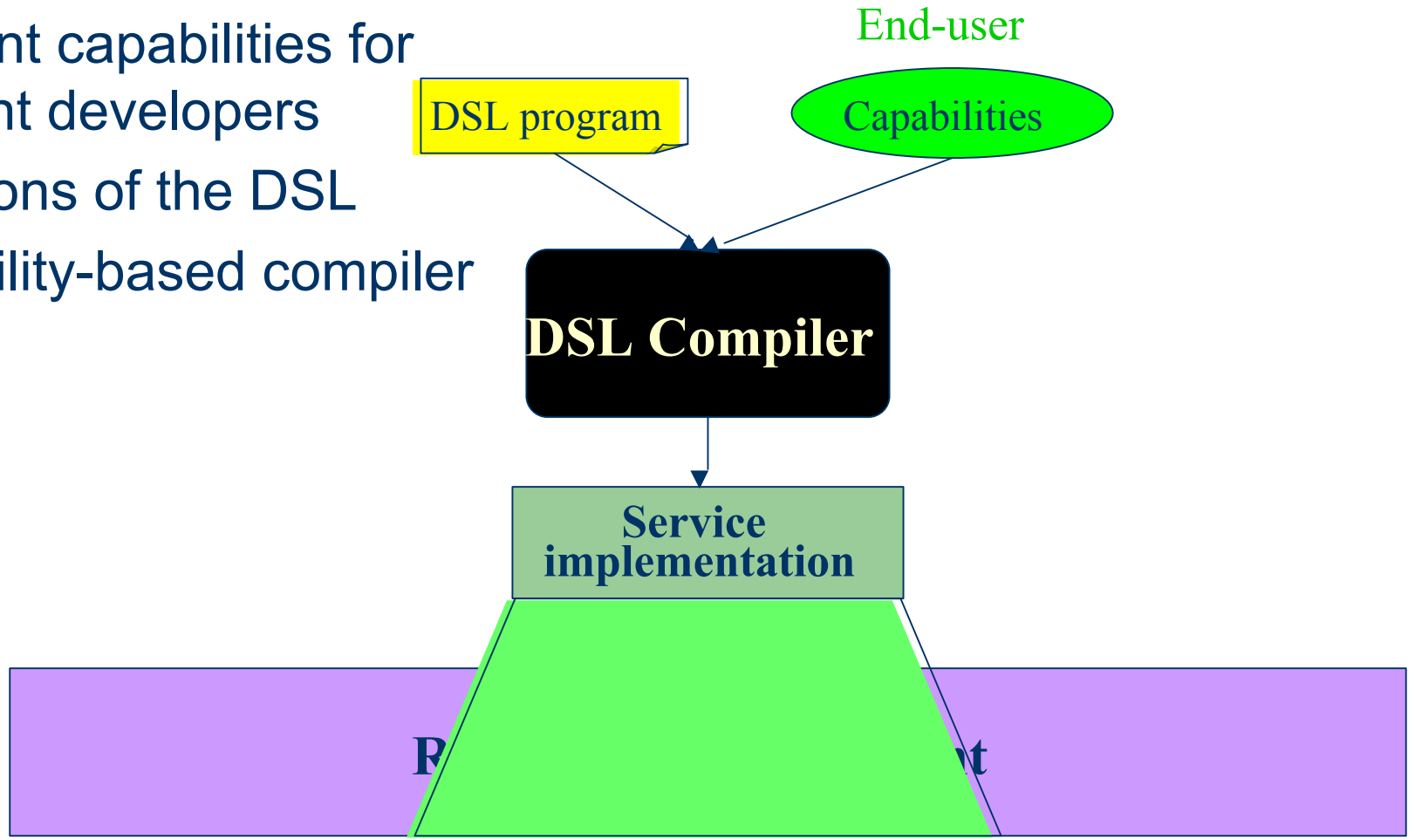
Multi-Level Programming

- Different capabilities for different developers
- Variations of the DSL
- Capability-based compiler

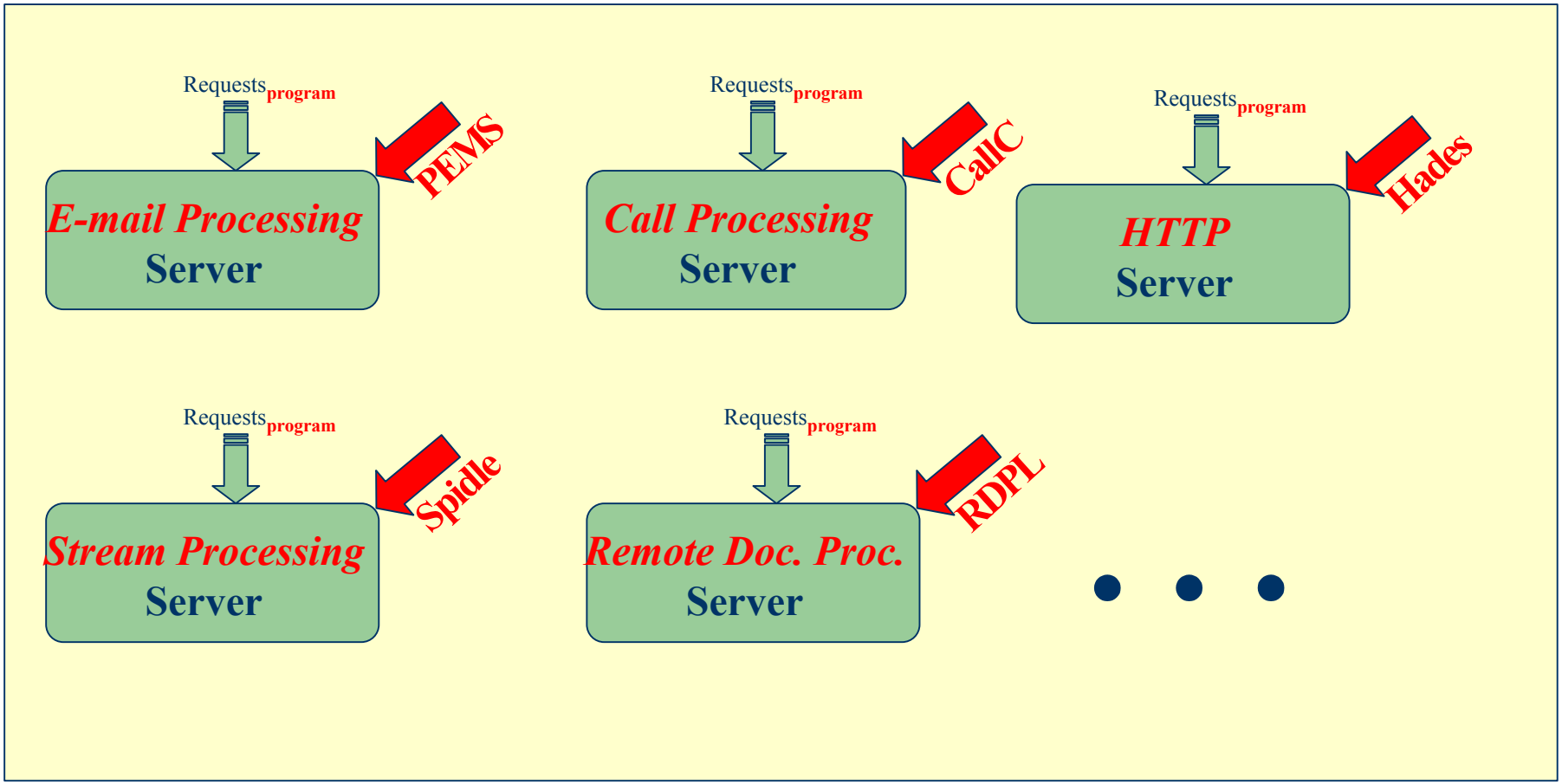


Multi-Level Programming

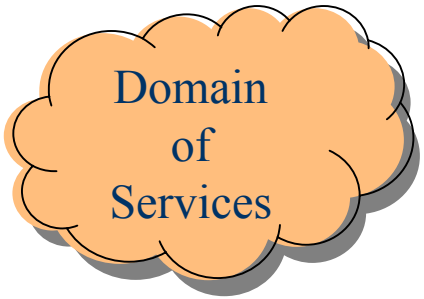
- Different capabilities for different developers
- Variations of the DSL
- Capability-based compiler

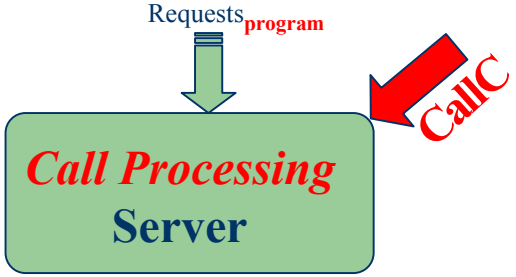


The Nova Platform: Functional Decomposition



Domain of Services: *Call Processing*



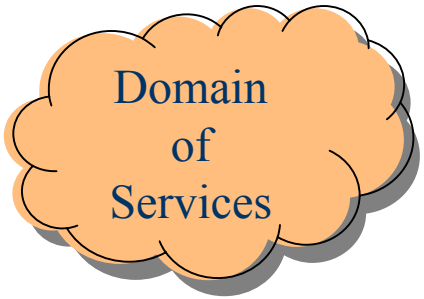


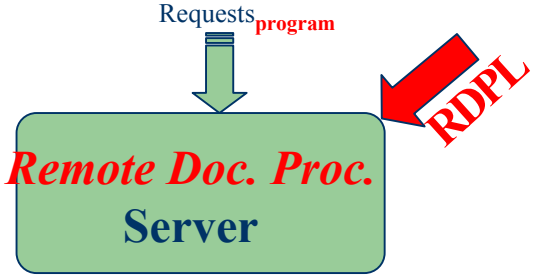
- Variations:
  - Call services
  - Web applications

```
incoming(call in)
{
  location l;

  switch proxy(sip:dana@labri.fr)
  {
    rediction(l):
      if(! match(l::host,"voicemail"))
      {
        redirect l;
        break;
      }
    default:
        proxy(sip:john@voicemail.labri.fr);
  }
}
```

Domain of Services: *Remote Document Processing*





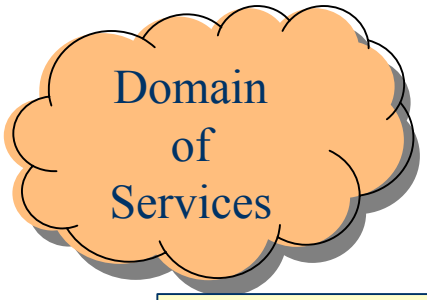
- Variations:
  - Sink definition
  - Conversion rules

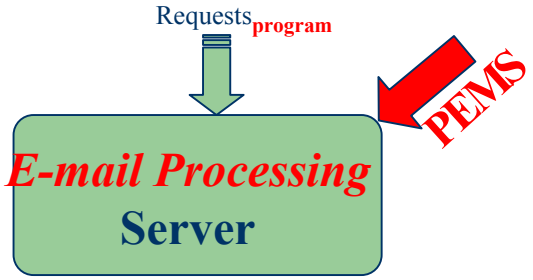
```
SINK d2r2
HLP This_is_the_printer_on_the_third_floor
CAPA Overleaf                boolean
IFACE /usr/bin/d2r2
SPOOL /var/spool/sinkd/d2r2
FORMAT ps
```

```
FORMATS gif jpeg ps ipq plain pdf

CONVERT plain ps /usr/local/transform/enscript
CONVERT jpeg ipq /usr/local/transform/jpgipq
CONVERT gif ps /usr/local/transform/gif2ps
CONVERT jpeg gif /usr/local/transform/jpeg2gif
CONVERT ps pdf /usr/local/transform/ps2pdf
```

Domain of Services: *E-mail Processing*

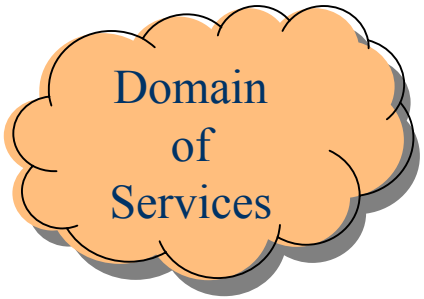


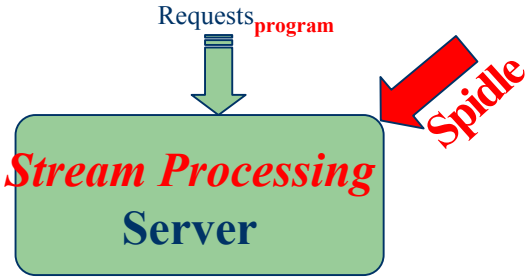


- Variations:
  - E-mail services
  - Remote doc. processing

```
view accesspoint PDA {
 ...
 Mailbox_view = nomadic(1MB);
}
view mailbox nomadic(size s) {
  if(From == "joe@mail.fr") bind boss;
  if(Size > s) ignore;
  ...
}
view message boss {
 From as cst("The Boss!");
 Date;  Subject;  Body;
 Attachment[] as bwImages(30KB);
}
view field Attachment bw(size s) {
 if(Attachment.size > s)
   return "too big" + Attachment.size;
 ...
}
```

Domain of Services: *Stream Processing*

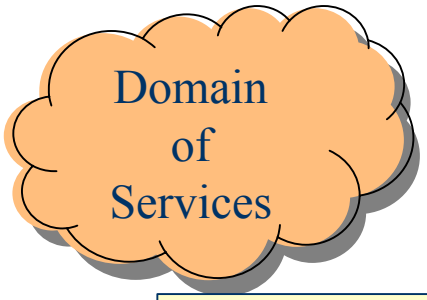


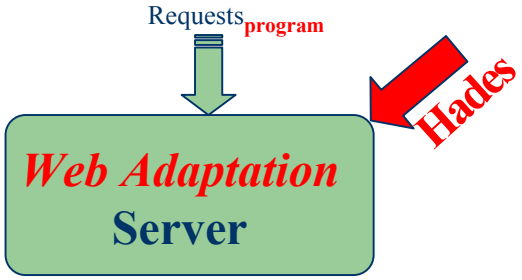


- Variations:
  - Stream processing
  - Composition

```
merger Frame_Merger {
  interface {
   stream in bit[7] nc;
   stream in bit[2] bc;
   stream in bit[2] Mc;
   stream in bit[13][3] xMc;
   stream in bit[6] xmaxc;
   stream out bit[56] bits;
  }
  map {
   nc # bc # Mc # xmaxc # xMc -> bits;
  }
}
```

Domain of Services: *Adaptation of Web resources*

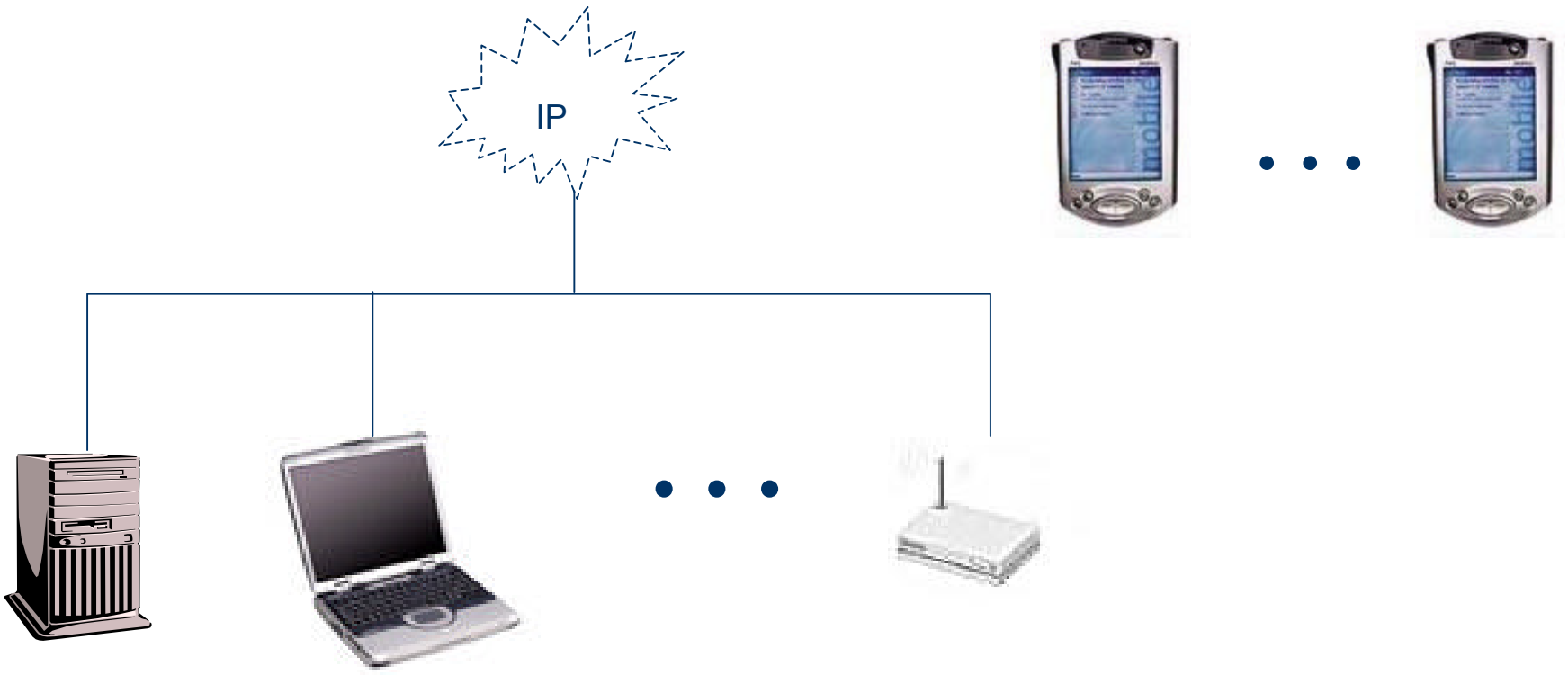




- Variations:
  - Web resource processing
  - Composition

```
Accesspoint PDA_meteo {
  html {
    title {
      setTitle(name + "(with hades)");
      keep();
    }
    element {
      keep();
    }
    image {
      if ((name == "meteo/france.jpg")
        keep();
      else
        remove();
    }
    text {
      keep();
    }
}
```

A System Architecture For the Nova Platform



Ongoing/Future Projects (1): Service Description Languages

- Multi-level programming
  - Capability models
  - Capability-based analysis
  - Capability-based compilation
- Service cost
  - Resource usage
  - Cost model
  - Composing models

Ongoing/Future Projects (2): Programmable Servers

- Evaluation studies
  - Robustness/memory usage/bandwidth usage/…
  - Individual servers
- Methodology for programmable servers
  - Making a server programmable
  - Proving correctness of a programmable server