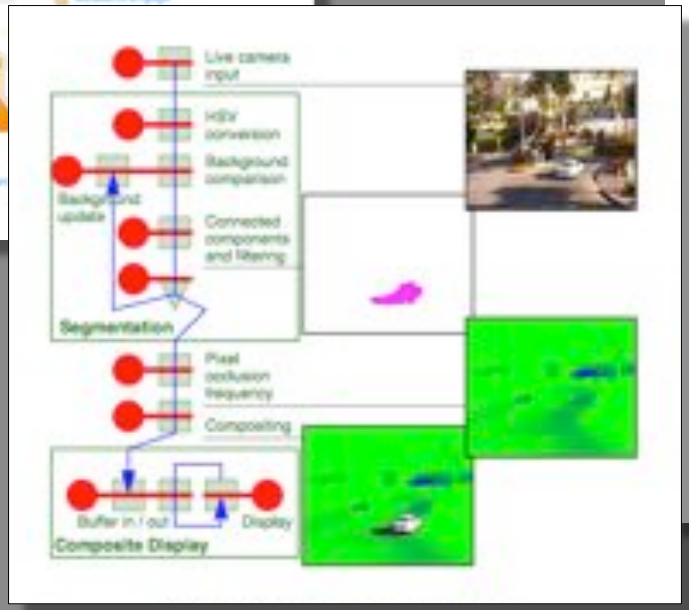
The Thorn Programming Language: Robust Concurrent Scripting

IBM Research	Purdue	Stockholm University	Texas, Arlington	Cambridge
Bard Bloom	Brian Burg	Johan Östlund	Nate Nystrom	Rok Strniša
Jakob Dam	Peter Maj	Tobias Wrigstad		
John Field	Gregor Richards			
	Jan Vitek			

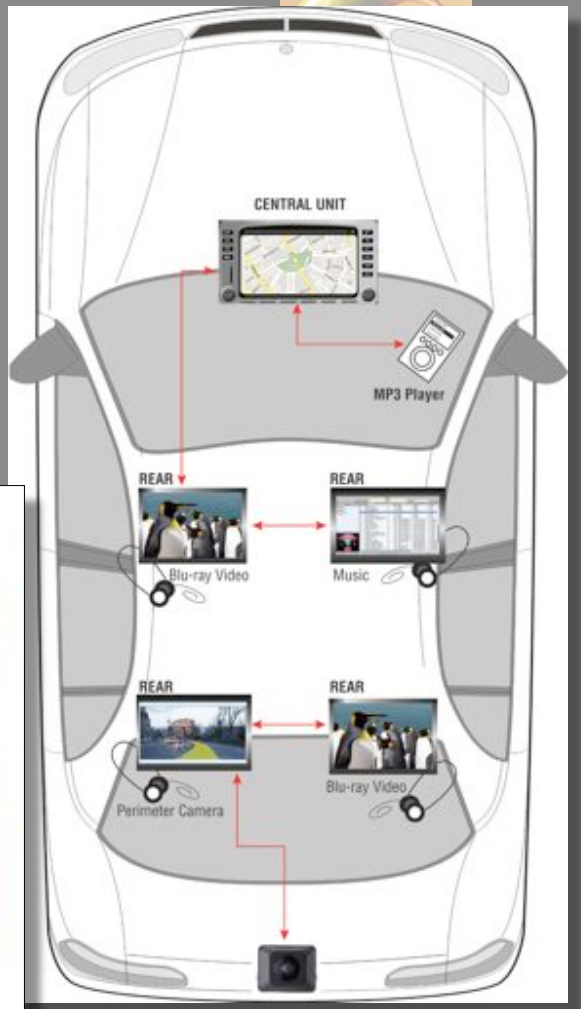
Java Languages Summit 2010

Do these apps have anything in common?

cloud-based web 2.0



real-time data analysis

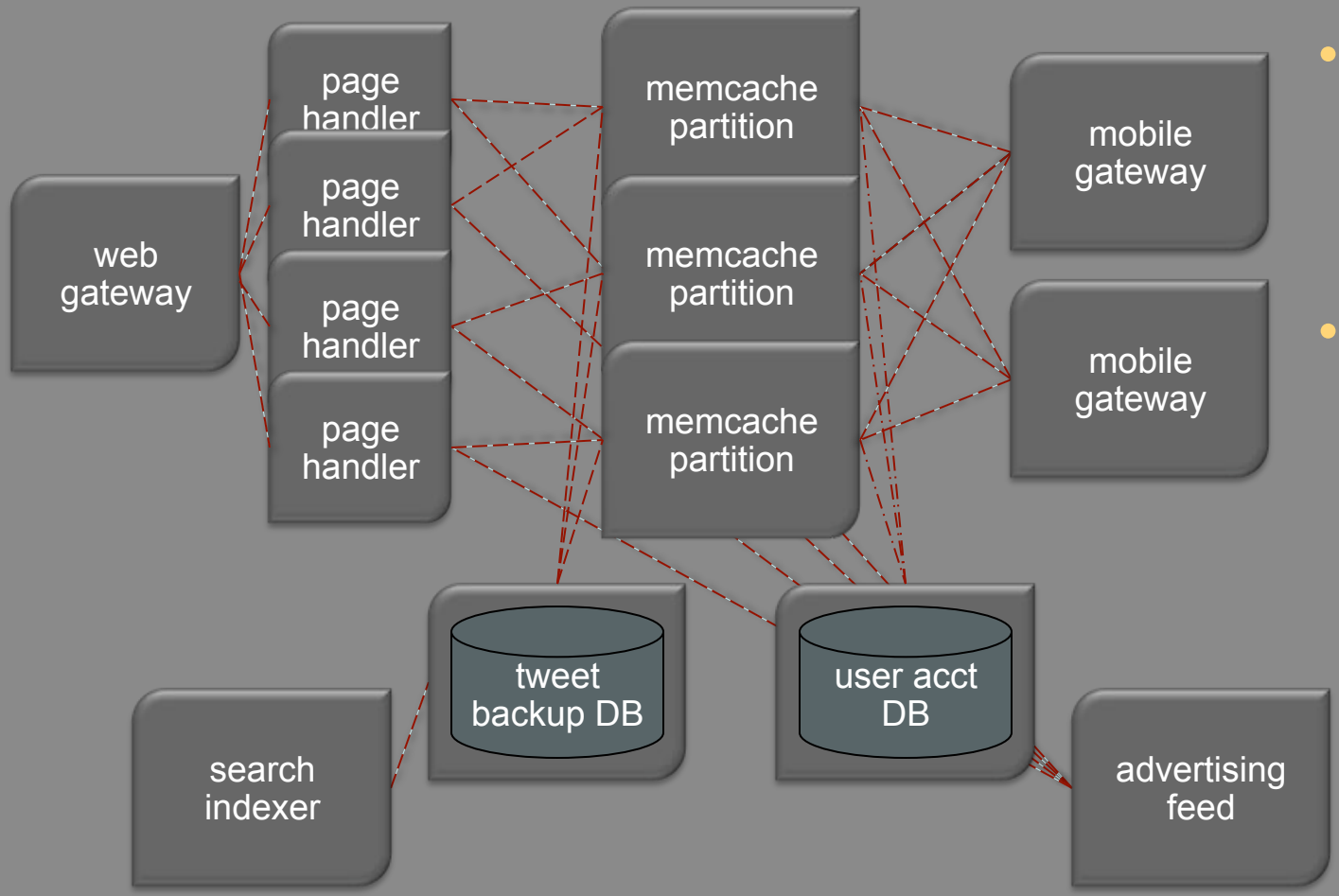


embedded network

Yes

- Collection of distributed, concurrent components
- Components are loosely coupled by messages, persistent data
- Irregular concurrency, driven by real-world data (“reactive”)
- High data volumes
- Fault-tolerance important

Example: Twitter



- each solid box is a logical process / event handler
- each dashed line is a message

Thorn goals

An open source, agile, high performance language for concurrent/distributed applications and reactive systems

Key research directions

- *Code evolution:* language, runtime, tool support for transition from prototype scripts to robust apps
- *Efficient compilation:* for a dynamic language on a JVM
- *Cloud-level optimizations:* high-level optimizations in a distributed environment
- *Security:* end-to-end security in a distributed setting
- *Fault-tolerance:* provide features that help programmers write robust code in the presence of hardware/software faults

Features, present and absent

Features

- isolated, concurrent, communicating processes
- lightweight objects
- first-class functions
- explicit state...
- ...but many functional features
- powerful aggregate datatypes
- expressive pattern matching
- dynamic typing
- lightweight module system
- JVM implementation and Java interoperability
- gradual typing system (experimental)

Non-features

- changing fields/methods of objects on the fly
- introspection/reflection
- serialization of mutable objects/references or unknown classes
- dynamic code loading

Status

- Open source: http:\\www.thorn-lang.org
- Interpreter for full language
- JVM compiler for language core
  - no sophisticated optimizations
  - performance comparable to Python
  - currently being re-engineered
- Initial experience
  - web apps, concurrent kernels, compiler, ...
- Prototype of (optional) type annotation system

Simple Thorn script

```
for (l <- argv()(0).file().contents().split("\n"))
  if (l.contains?(argv()(1))) println(l);
```

- access command-line args
- file i/o methods
- split string into list
- iterate over elements of a list
- no explicit decl needed for var
- usual library functions on lists

grep

DEMO

More complex example: a MMORPG*

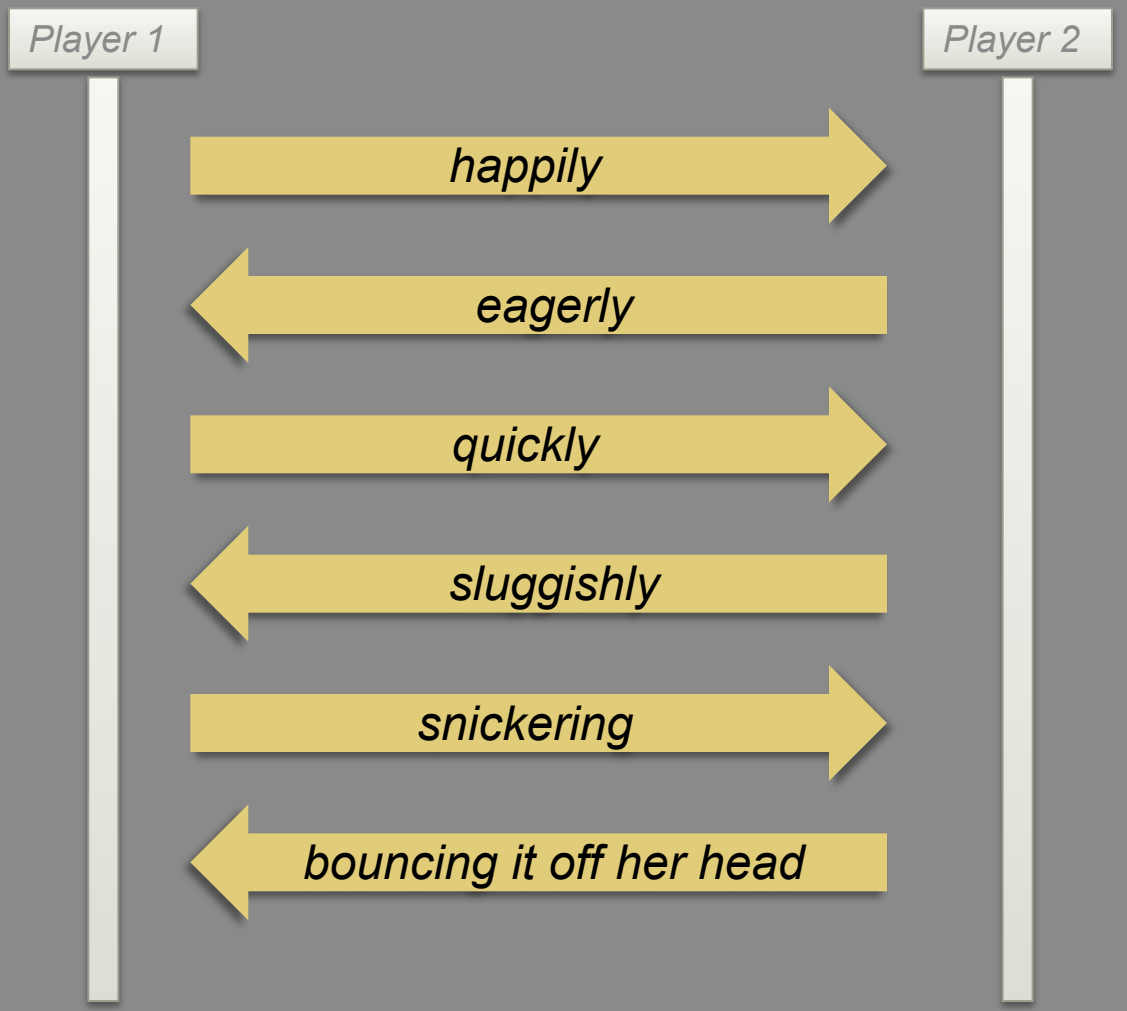
- Adverbial ping-pong
- Two players
- Play by describing how you hit the ball
- Distributed
- Each player runs exactly the same code

*minimalist multiplayer online role-playing game

MMORPG

DEMO

MMORPG message flow



MMORPG code

```
// MMORPG code for both player

spawn {

  var done := false;

  body {
    [name, otherURI] = argv();
    otherSite = site(otherURI);

    fun play(hit) {
      advly = readln("Hit how?");
      done := advly == "";
      if (done) {
        println("You lose!");
        otherSite <<< null;
      }
      else {
        otherSite <<<
        "$name $`hit`s the ball $advly.";
      }
    }

  start =
    thisSite().str < otherSite.str;

  if (start) play("serve");

  do {
    receive {
      msg:string => {
        println(msg);
        play("return");
      }
    | null => {
        println("You
        done := true;
      }
    }
  } until (done);
}
```

spawn an isolated *component* (process)

mutable component-scoped variable

immutable component-scoped variable

convert URI into component ref

function decl

receive messages matching *pattern*

send a message (any immutable datum)

pattern variable (with type constraint)

constant pattern

interpolate data into string

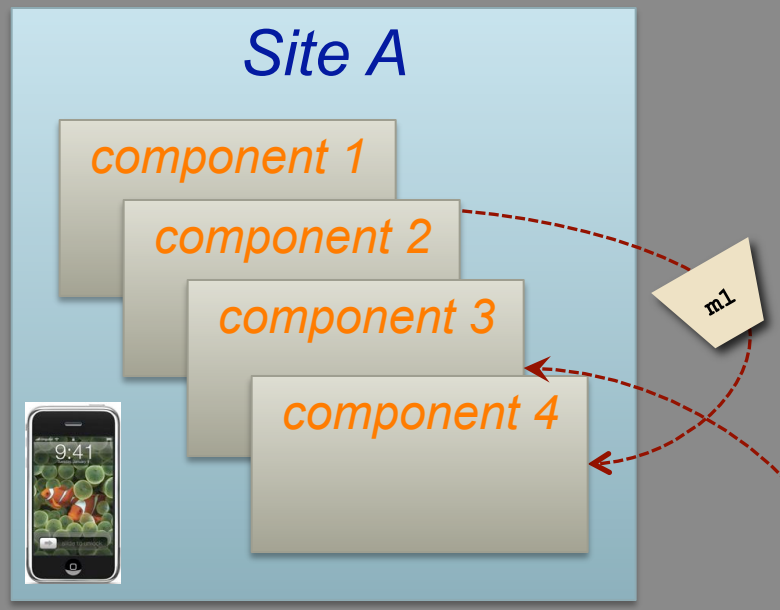
Thorn design philosophy

- Steal good ideas from everywhere
  - (ok, we invented some too)
  - aiming for harmonious merge of features
  - strongest influences: Erlang, Python (but there are many others)
- Assume concurrency is ubiquitous
  - this affects *every* aspect of the language design
- Adopt best ideas from scripting world...
  - dynamic typing, powerful aggregates, ...
- ...but seduce programmers to good software engineering
  - powerful constructs that provide immediate value
  - optional features for robustness
  - encourage use of functional features when appropriate
  - no reflective or self-modifying constructs
- Syntax follows semantics
  - more consequential ops have heavier syntax

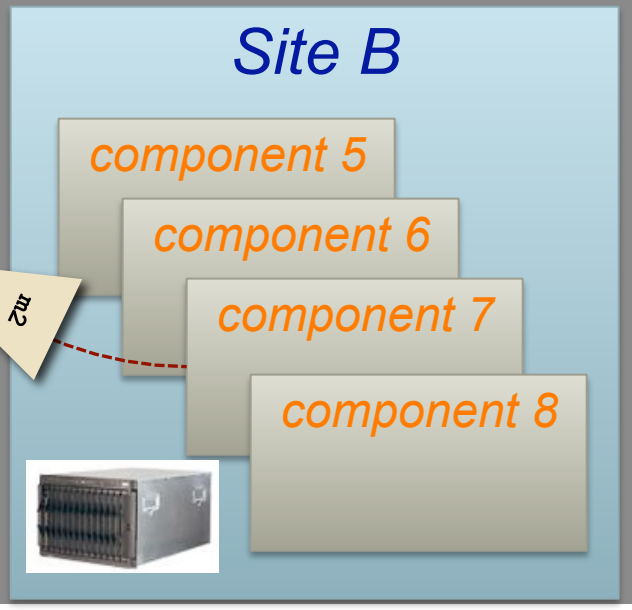
Scripting + concurrency: ? …or… !

- Scripts already handle concurrency (but not especially well)
- Dynamic typing allows code for distributed components to evolve independently…code can bend without breaking
- Rich collection of built-in datatypes allows components with minimal advance knowledge of one another's information schemas to communicate readily
- Powerful aggregate datatypes extremely handy for managing component state
  - associative datatypes allow distinct components to maintain differing "views" of same logical data

Thorn app: birdseye view



- *components* are Thorn processes
- components can spawn other components (at the same site)
- processes communicate by message passing
- intra- and inter-site messaging *works the same way*



- *sites* model physical application distribution
- one JVM per site
- I/O and other resources managed by sites
- failures managed by sites

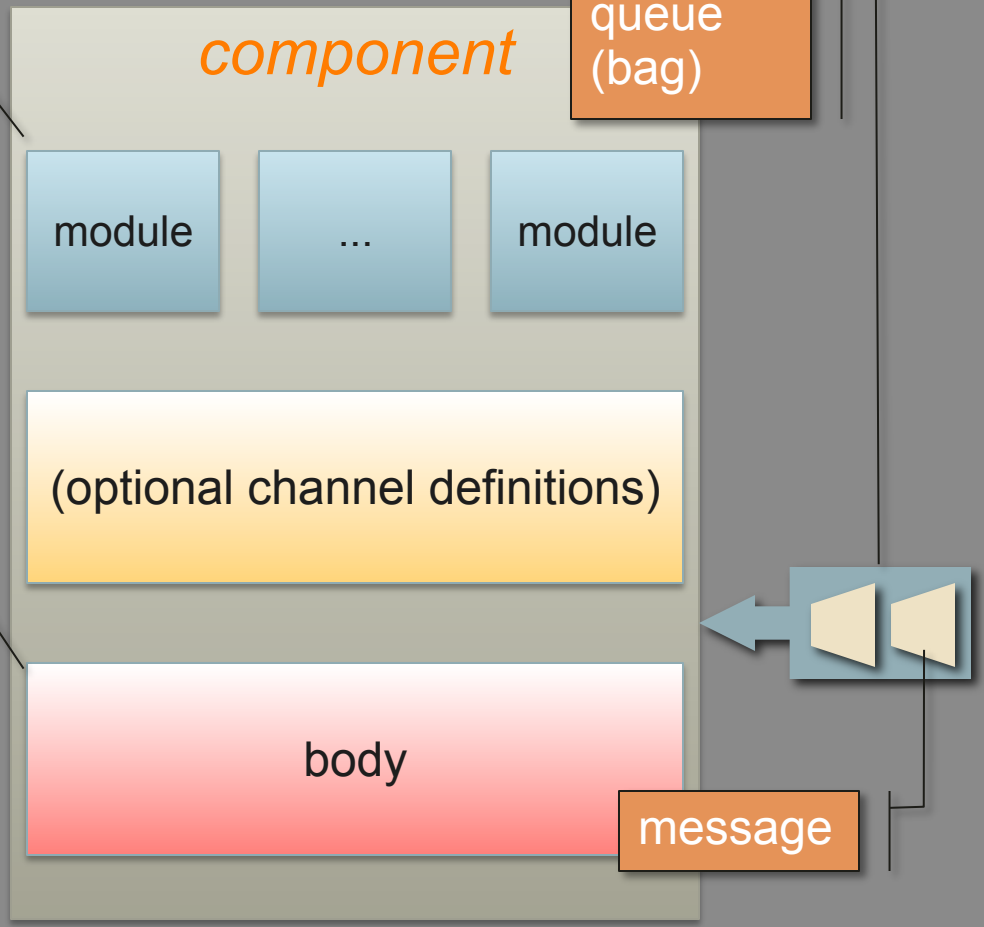
Anatomy of a component

- defines the component's code and state
- loaded and initialized when component is spawned

- all program state encapsulated in one or more components
- each component has a *single thread of control*
- components are *isolated*
  - no shared state
  - exceptions do not propagate across components
- messages passed by *value*
- *compartments* and refs are *only* form of remote reference

- statement executed when component is spawned (usually a loop)
- component execution ends when body ends

- messages managed via a simple "mailbox" queue
- no locks (in user code)



Modules

- modules are reusable bundles of *definitions*
  - of code
    - functions
    - classes
  - of state
    - mutable
    - immutable
- modules may import other modules
- modules loaded (only) when enclosing component is spawned
- set of modules used by any component defined *statically*
- modules are initialized once, on first import

(selectively) import definitions from other modules

bind immutable state variable

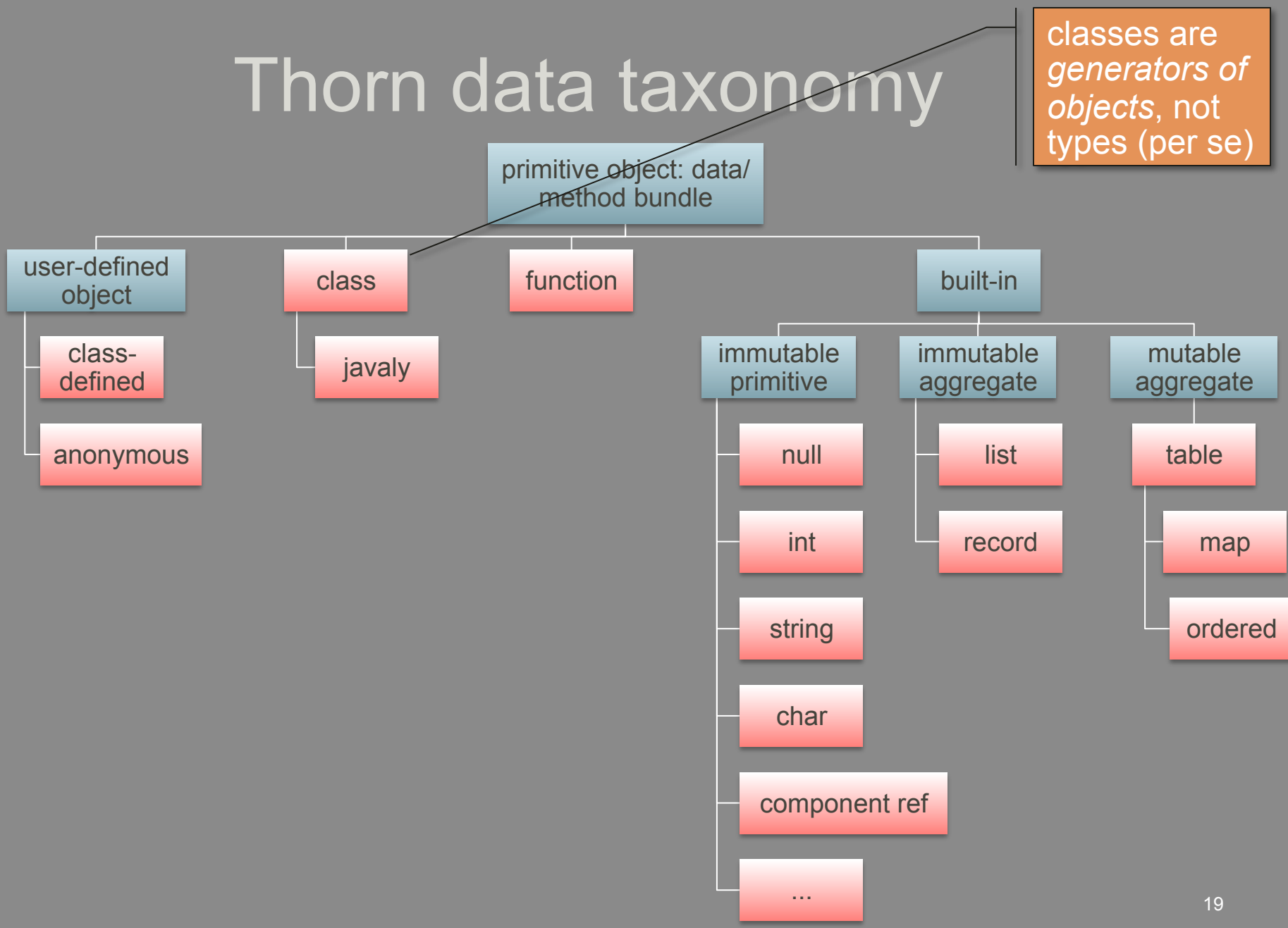
initialize mutable state variable

define class (which may access values of previous definitions)

define function (here, overloaded)

```
module webhtml;

import webschemas.html1strict;

enforce_validity = true;
var xhtmlchecker := null;

class Element(
  var content:list, attr, type) {

  def str() =
    "".join(%[ c.str | for c <- content ]);
  ...
}

fun head(args, content) =
      Element(args, content, "head");
  | head(content) =
      Element({: :}, content, "head");
```

Thorn data taxonomy



More robust scripting

- No reflection, eval, dynamic code loading
  - alternatives for most scenarios
- Ubiquitous patterns
  - for documentation
  - to generate efficient code
- Powerful aggregates
  - allow semantics-aware optimizations
- Easy upgrade path from simple scripts to reusable code
  - simple records → encapsulated classes
- Channel-style concurrency
  - to document protocols
- Modules
  - easy to wrap scripts, hide names
- Experimental gradual typing system

Thorn patterns

match *value* of `k`

declare and bind variable `y`

"I found it, and it's `y`!"

```
alist = [ [1, true], [15, null], ["yes", "no"] ];

fun lookup(k, [ [$(k), v], _... ]) = +v;
  | lookup(k, []) = null;
  | lookup(k, [_, t...]) = lookup(k, t);

if ( lookup(15, alist) ~ +w ) // found it
```

match arb. tail

"I didn't find it"

idiom for "did you find something (call it `w`)?"

Patterns are everywhere

- **fun** f(*Pat1* ... Patn)
- *Pat* = *Exp*
- **match**(*Exp*) {*Pat1* ... *Patn*}
- **receive** {*Pat1* ... *Patn*}

Lists, queries

random number in 1 to `nSides`

list method (nullary, hence may omit parens)

```
fun roll(nDice, nSides) =
  %[ nSides.rand1 | for i <- 1 .. nDice ].sum;
```

- `%[` *Z* `|` **`for`** `i <-` *E* `]`
  - list of the values of *Z* varying `i`
  - this one makes a list of random numbers

Records and tables

key

values

`var` = conveniently update one field "in place"

```
chirps = table(num){chirp; var plus, minus};
  ...
chirps(n) := {: chirp:c, plus:p, minus:m :}
```

update row with key `n` (other ops check if row already exists)

- Tables are high power maps/dictionaries
- Each row of a table is a *record*
- Can add/delete rows
- Adding a new column is easy; no need for objects or parallel tables
- Variants: *ordered* (extensible arrays), *map*-style
- Wide selection of queries

Records to objects

- Prototype with records

```
r = {: a:1, b:2 :}
```

- Upgrade later to classes

```
class Abc(a,b) { def aplusb() = a + b; };
  ...
r = Abc(1, 2);
```

- And things still work
  - access via selectors

```
r.b == 2
```

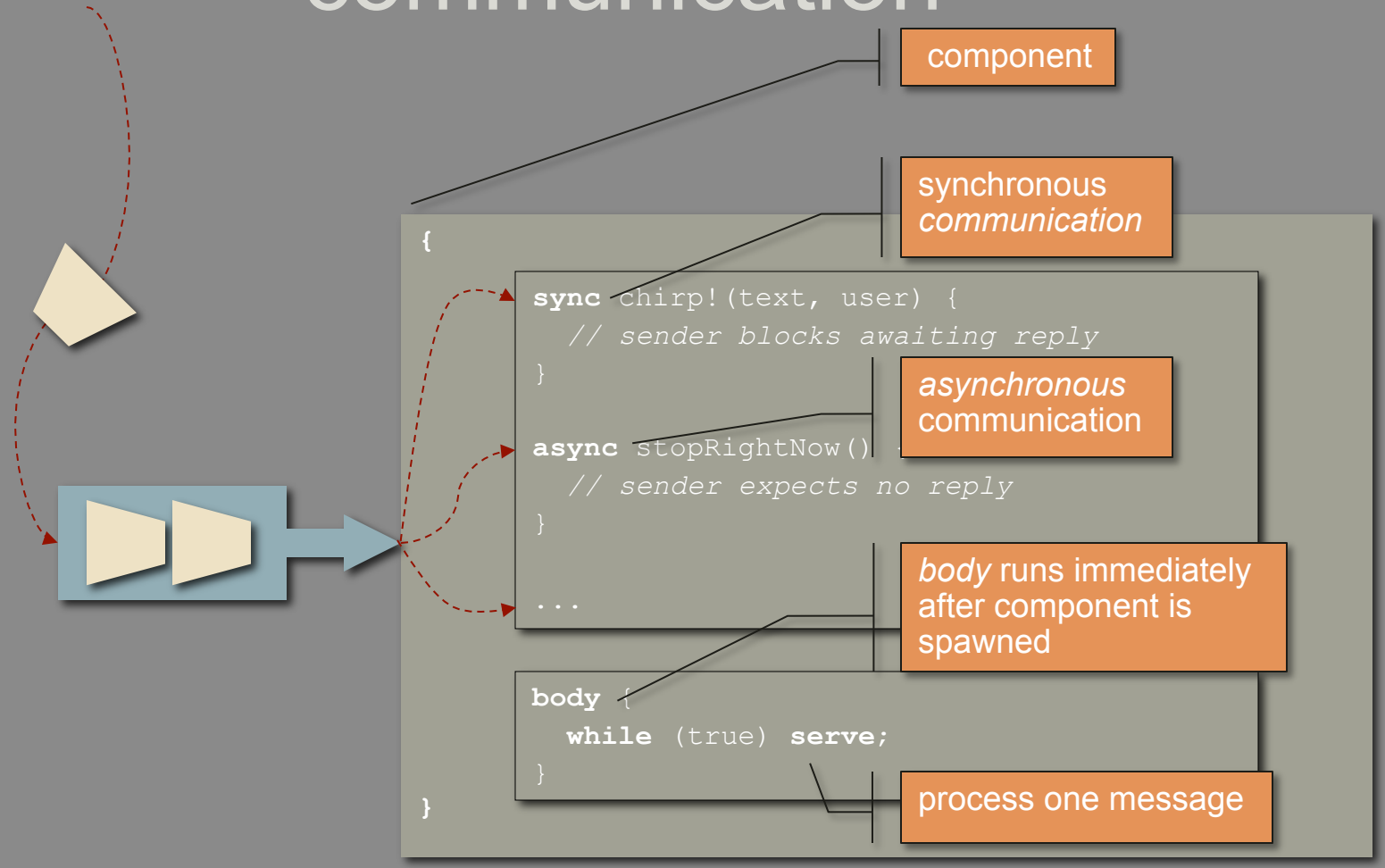
  - access via pattern matching

```
if (r ~ {: a :}) println(a);
```

- Plus, you get method calls

```
r.aplusb() == 3
```

Channel-style communication



Channels are sugar on basic messaging primitives

Compiling Thorn to the JVM

Message dispatch

- compiler generates a Java Interface per method signature (name/arity)
- a Thorn class is compiled to a Java class that implements as many interfaces as it has methods
- dispatch compiles to a cast operation following by an interface dispatch
- number of interfaces can be reduced by grouping methods together in batches

Compiling Thorn to the JVM

Fields

- every Thorn field access is compiled to a Java method call
- all fields are compiled to private fields in Java
- all inherited fields are re-declared in each generated Java class
- setter methods for val fields throw exceptions

Optimizing Thorn

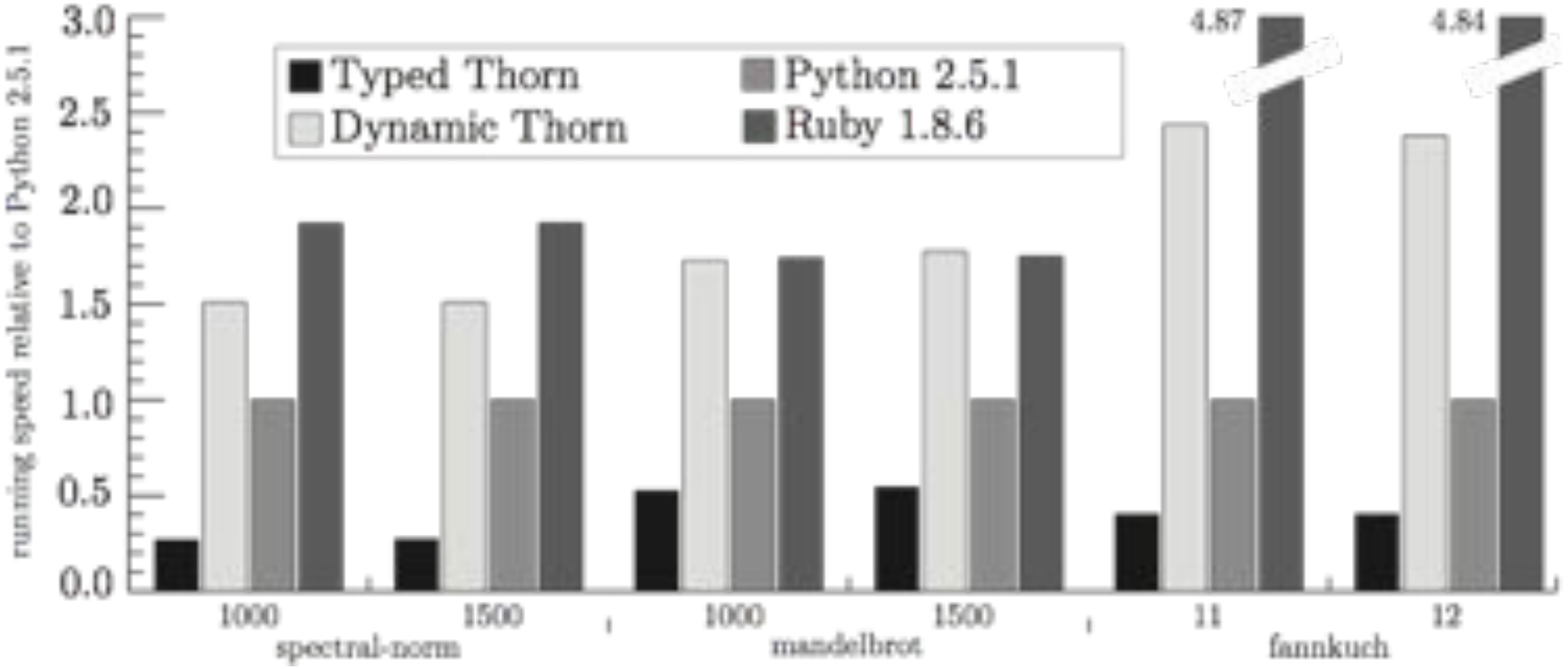
```
fun a(i, j) =
  1.0 / (((i + j) * (i + j + 1) >> 1) + i + 1);
```

- 87 bytecode instructions, 8 new frames, 8 new objects

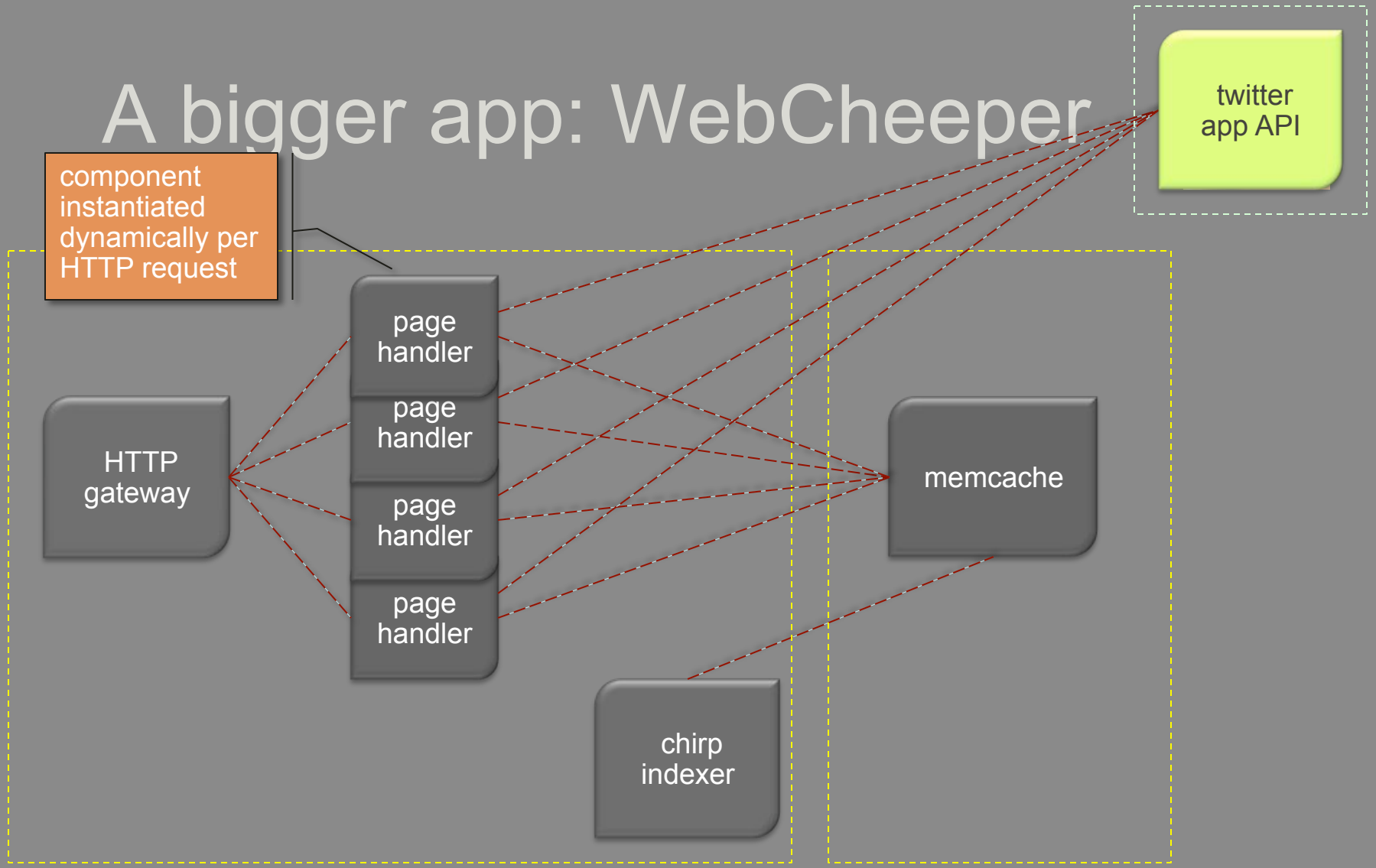
```
fun a(i: int, j: int) =
  1.0 / (((i + j) * (i + j + 1) >> 1) + i + 1);
```

- 29 bytecode instructions, 0 new frames, 1 new object (because of untyped return)

Performance



A bigger app: WebCheeper

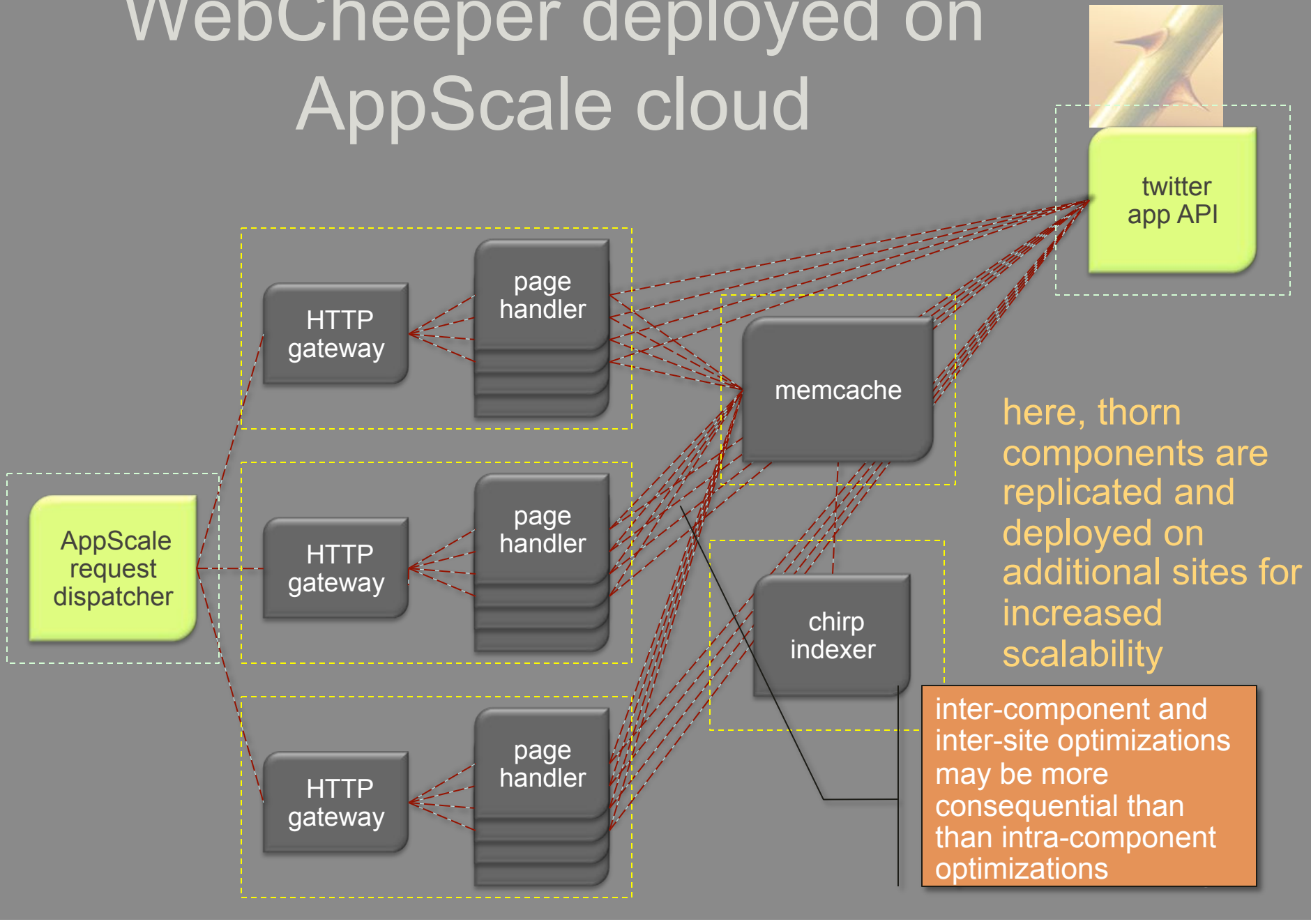


- each solid box is an isolated Thorn component
- each dashed box is a Thorn site

WebCheeper

DEMO

WebCheeper deployed on AppScale cloud



Thorn: research testbed

In progress

- optimizing compiler
- cloud-level optimizations
  - code, data placement
  - serialization
  - message piggybacking
- component-level security
  - information flow
  - access control
- join-style patterns for synchronization
- database integration

Planned

- lighter-weight Java integration
- refactored, componentized libraries
- much more work on optional types
  - e.g., generalization of patterns
- failure recovery for components
- static checkers

More information

- http://www.thorn-lang.org
  - download interpreter
  - links to papers
  - online demo
- Additional collaborators welcome!
- Workshop (NB: moved to Wed, 11:30)
  - larger code examples
  - more on
    - concurrency
    - tables/queries
    - classes
    - optional types
    - Java interop
    - JSON/HTTP handling
  - some lessons learned
  - discussion: role of languages in distributed/cloud apps

Questions?