

# Protocol Analysis

17-654/17-754

Analysis of Software Artifacts

Kevin Bierhoff

# [ Take-Aways ]

---

- Protocols define temporal ordering of events
  - Can often be captured with state machines
- Protocol analysis needs to pay attention to
  - Interprocedural control flow
  - Aliasing of objects
- Disjoint sets and capabilities can handle aliasing correctly
  - Fractional permissions for heap sharing
- State changes correspond to field changes

# [ Agenda ]

---

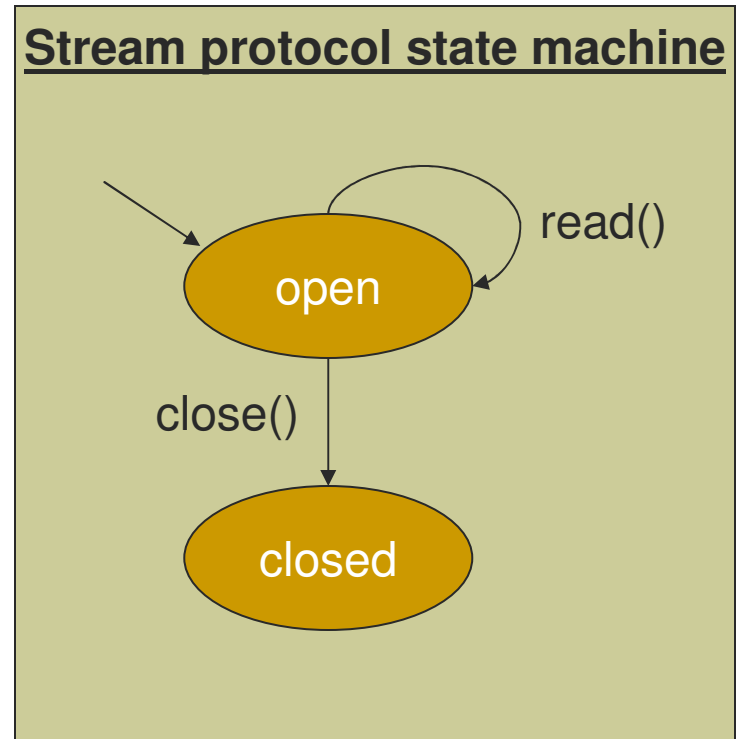
- ➔ Example protocols ←
- Modeling protocols as state machines
  - Protocol analysis approaches
    - Annotations vs. interprocedural analyses
  - Aliasing challenges
    - Tracking aliases in methods and fields
  - Protocol implementation checking

# Streams can be read until they're closed

```
public interface InputStream {  
    public int read();  
    public void close();  
}
```

## Stream sample client

```
InputStream f = new FileInputStream(...);  
int c = f.read(); // read first character  
while(c >= 0) {  
    // do something with c  
    c = f.read(); // read next character  
}  
f.close();
```



# Sockets go through a well-defined sequence of states

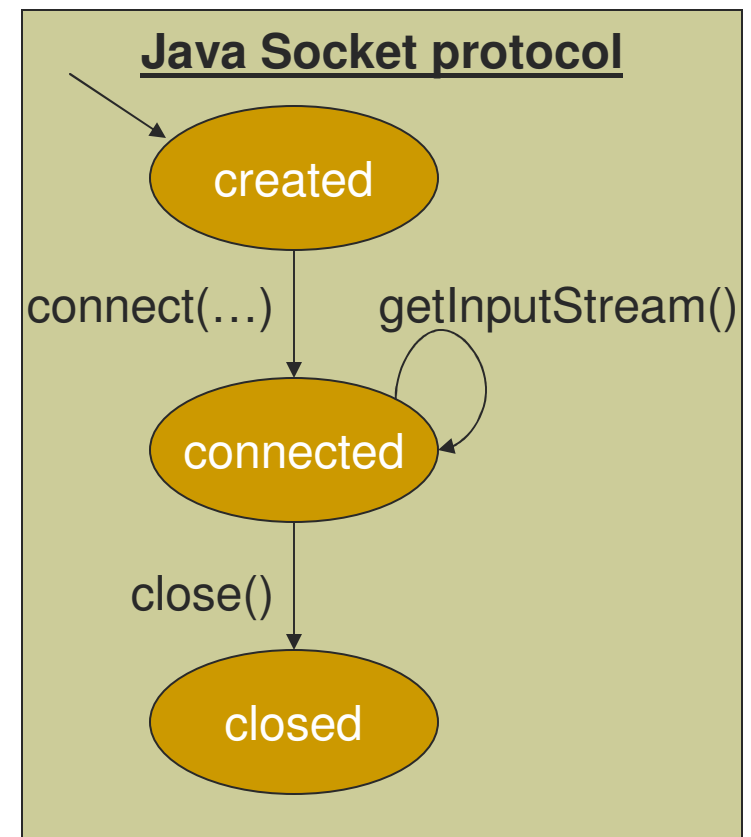
```
@States({"created", "connected", "closed"})
public class Socket {
    @Creates("created")
    public Socket()

    @ChangesState("created", "connected")
    public void connect(...)

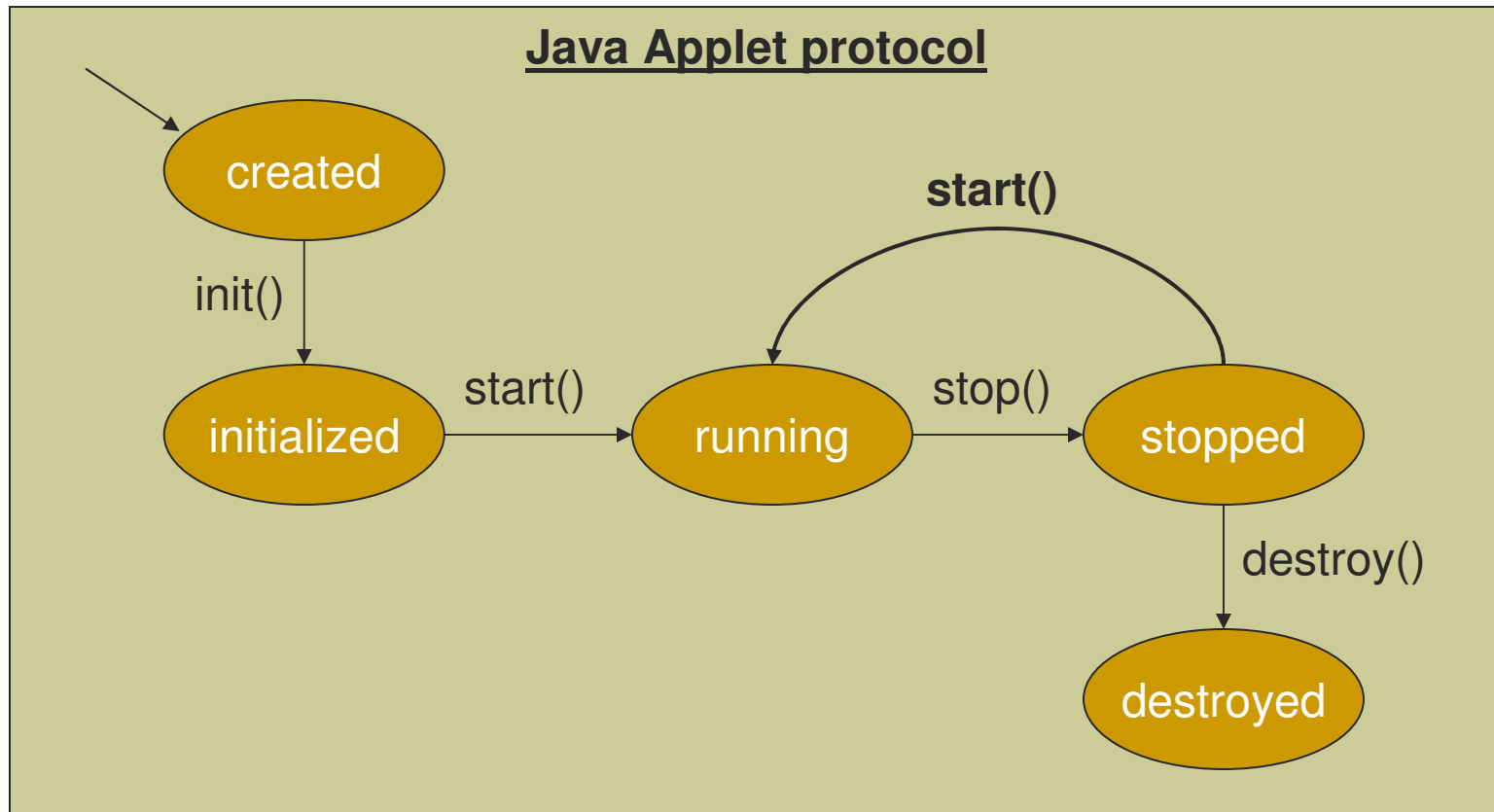
    @InState("connected")
    public InputStream getInputStream()

    @InState("connected")
    public OutputStream getOutputStream()

    @ChangesState("connected", "closed")
    public void close();
}
```

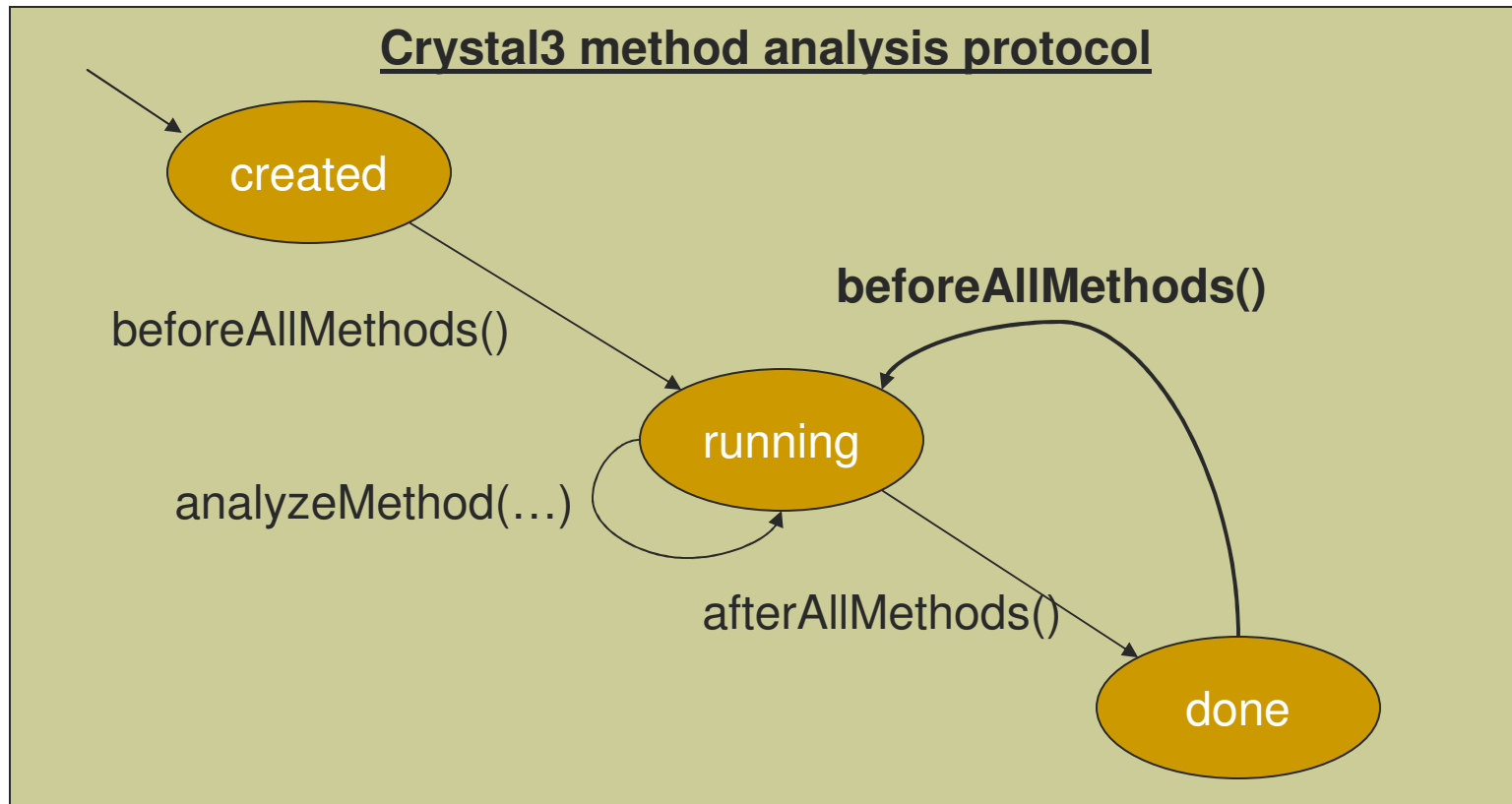


# Java Applets have a funny back edge



Example based on: G. Fairbanks, D. Garlan & W. Scherlis. Design fragments make using frameworks easier. In *Proceedings of OOPSLA'06*, pp. 75-88. ACM Press, 2006.

# Crystal3 analyses have the same back edge



Unawareness of this back edge can lead to outdated error reports

# Protocols constrain temporal ordering of events

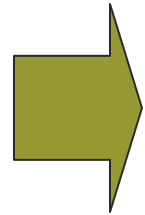
- Protocols define restrictions on which methods can be called when
- Clients have to follow protocols in order to avoid runtime errors
- Protocols can often be modeled as **state machines**



# [ Protocol documentation... ]

- Protocols are informally documented
  - Example: `java.io.InputStream`
    - Detailed Javadoc for every method
  - Example: `java.net.Socket`
    - Exceptions describe when methods *cannot* be called
- Not always complete and precise

# [ ...formalized in various ways ]



Formalization	Socket example
Annotations on classes and methods	<pre>@States({"created", "connected", "closed"}) public class Socket {     @Creates("created") public Socket()     @ChangesState("created", "connected")     public void connect(...) ...</pre>
Regular expressions	<pre>connect (getInputStream   getOutputStream)* close</pre>
State machine defined in one place (similar to Metal)	<pre>created : connect(...) -&gt; connected connected :     getInputStream() -&gt; connected       close() -&gt; closed</pre>

**We will use annotations on classes and methods**

# [ Agenda ]

---

- Example protocols

- Modeling protocols as state machines

➔ Protocol analysis approaches ←

- Annotations vs. interprocedural analyses

- Aliasing challenges

- Tracking aliases in methods and fields

- Protocol implementation checking

# Protocol analysis tracks states of variables

Socket sock = <b>new</b> Socket();	<u>Post-state</u> Created
sock.connect( <b>new</b> InetAddress("www.cs.cmu.edu",80));	Connected
InputStream in = sock.getInputStream();	Connected
sock.close();	Closed

- What if *sock* is assigned to another variable?
- What if *sock* is assigned to a field?
- ➡ What if *sock* is passed to another method? ←

# Calling other methods

```
public class SocketClient {  
  
    private String readSocket(Socket s) {  
        InputStream in = s.getInputStream();  
        ... // read and return string  
    }  
  
    public String readRemoteData() {  
        Socket sock = new Socket();  
        sock.connect(new InetSocketAddress(  
            "www.cs.cmu.edu", 80));  
        String result = readSocket(sock);  
        sock.close();  
        return result;  
    }  
}
```

Is this call ok?

Is this call ok?

Need to handle inter-procedural control flow

# Interprocedural analysis techniques

- Need to handle inter-procedural control flow
  - Every method call could potentially affect analysis results
  - Need to figure out what happens in called methods
- Some possible approaches
  - Default assumptions
  - Interprocedural CFG
  - More annotations

# Defaults too inflexible for protocol analysis

- Simple approach: **default assumptions**
  - Assumption about method parameters and result
  - Check that call and return sites respect the default
  - Example: Maybe-null assumption in null analysis
    - Assume that method parameters may be null
    - Check methods with that assumption
    - All call and return sites automatically maybe-null
- No reasonable default for protocol analysis
  - “Any” state too imprecise (lots of false positives)
  - Optimistic assumption (a particular state) might be wrong a lot of the times

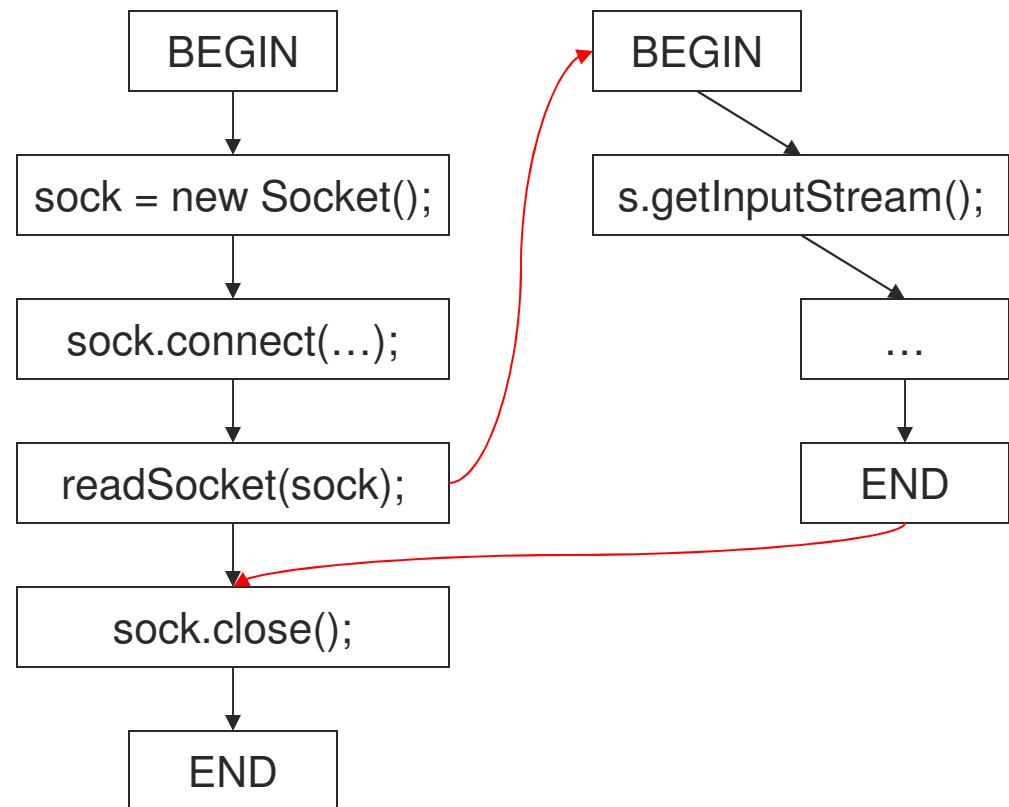
# Interprocedural CFG “inlines” method calls

## Interprocedural CFG

- Pretend that called methods are part of current method
- Every method appears once

Problem: scalability

- One big CFG for the entire program



**Interprocedural CFG hard to use at scale**



# Assume and Check Annotations

```
String readSocket(  
    @InState("connected") Socket s) {  
    InputStream in = s.getInputStream();  
    ... }  
}
```

## ■ Annotations

- Starting dataflow value for all parameters
- Dataflow value for result

## ■ Verification

- Initial info: starting value for parameters
- Verify result  $\sqsubseteq$  annotation<sub>result</sub>
  - Ending value for result obeys annotation
- Verify arg  $\sqsubseteq$  annotation<sub>arg</sub>
  - Actual arguments obey annotations on formal parameter

# [ Agenda ]

---

- Example protocols
  - Modeling protocols as state machines
- Protocol analysis approaches
  - Annotations vs. interprocedural analyses
- ➡ Aliasing challenges ←
- Tracking aliases in methods and fields
- Protocol implementation checking

# Looks familiar? Aliasing is a problem that you can easily have

	<u>t1</u>	<u>t2</u>	<u>t3</u>
SimpleProtocolTest t1 = <b>new</b> SimpleProtocolTest();	a	--	--
SimpleProtocolTest t2 = <b>new</b> SimpleProtocolTest();	a	a	--
SimpleProtocolTest t3 = t1;	a	a	a
t1.aToB(); // t1 alias t3 in b, t2 in a	b	a	a
t1 = t2; // t3 in b, t1 alias t2 in a	a	a	a
t1.aToB();	b	a	a
t3.bToC();	b	a	ERR
t2.inB(); // t1 alias t2 in b, t3 in c	b	ERR	

Spurious warnings

**Aliasing = multiple names for the same thing**

# Track local aliases as disjoint sets (aka equivalence classes)

- Track aliased variables as disjoint sets
  - Lattice information
    - $A = \{ S_1, \dots, S_n \}$
    - $S_1, \dots, S_n$  disjoint sets of variables
  - Copy instructions  $x = y$ 
    - Get  $y$ 's aliases  $S \in A$  where  $y \in S$
    - Add  $x$  to  $S$  (and remove it from any other set)
  - Object allocations  $x = \text{new } C(\dots)$ 
    - Remove  $x$  from existing sets
    - $A = A \cup \{ x \}$  (i.e., add new set with just  $x$ )
    - (Need to also set initial state for  $x$ )
- Track state for each disjoint set
  - Method calls  $x = y.m(\dots)$ 
    - Get  $y$ 's aliases  $S = \{ y_1, \dots, y_n \}$  where  $y \in S$
    - Update  $S$ 's state according to  $m$ 's spec

# Disjoint sets correctly handle local aliases in example

	<u>aliasing</u>	<u>t1</u>	<u>t2</u>	<u>t3</u>
SimpleProtocolTest t1 = <b>new</b> SimpleProtocolTest();	{t1}	a	--	--
SimpleProtocolTest t2 = <b>new</b> SimpleProtocolTest();	{t1}, {t2}	a	a	--
SimpleProtocolTest t3 = t1;	{t1,t3}, {t2}	a	a	a
t1.aToB(); // t1 alias t3 in b, t2 in a	{t1,t3}, {t2}	b	a	b
t1 = t2; // t3 in b, t1 alias t2 in a	{t1,t2}, {t3}	a	a	b
t1.aToB();	{t1,t2}, {t3}	b	b	b
t3.bToC();	{t1,t2}, {t3}	b	b	c
t2.inB(); // t1 alias t2 in b, t3 in c	{t1,t2}, {t3}	b	b	c

States of aliased variables are updated correctly

# Calling other methods can affect fields

Our approach so far does not issue any warnings

```
public class AliasingFun() {  
    @InState("b") private SimpleProtocolTest t2;
```

```
private void callField() {  
    t2.inB();  
}
```

Field annotation makes this call go through

```
public void aliasingFun() {  
    SimpleProtocolTest t1 = ...  
    t1.aToB();  
    internal(t1);  
    t1.bToC();  
    callField();  
    ...  
}
```

t2 is actually in "c" when called

This call violates t2's annotation

```
private void internal(@InState("b") SimpleProtocolTest t) {  
    t2 = t;  
}
```

t2 aliases t and t1

Fields hold on to objects beyond duration of methods

# Aliasing through fields different from local variables

- Aliasing in local variables affects current method only
  - We can handle that with disjoint sets
- Fields hold on to objects
  - Assignment to field in one method can affect other methods
  - Changing state of local variable can inadvertently change state of field
- Other situations with similar problems?

# Capabilities track whether an object is accessible

- **Capabilities:** Access objects only if not stored in a field
- Exactly one capability for each object
  - Can call methods only if capability available
    - `x.m(...)` only valid if caller has capability for `x`
  - Capability created with **new**
  - Field assignments `x.f = y`
    - “Capture” capability for `y`
- Annotate methods with capabilities
  - `@Captured` if capability needed but not returned
  - `@Borrowed` if capability needed and returned



# Capabilities correctly handle field assignments and method calls

```
public class AliasingFun() {
    @InState("b") private SimpleProtocolTest t;

    private void callField() {
        t.inB();
    }
    public void aliasingFun() {
        SimpleProtocolTest t1 = new SimpleProtocolTest();
        t1.aToB();
        internal(t1);
        t1.bToC();
        callField();
        ...
    }
    private void internal(@Borrowed SimpleProtocolTest t) {
    }
    private void internal(@Captured SimpleProtocolTest t) {
        t2 = t;
    }
}
```

Error: No capability for t1

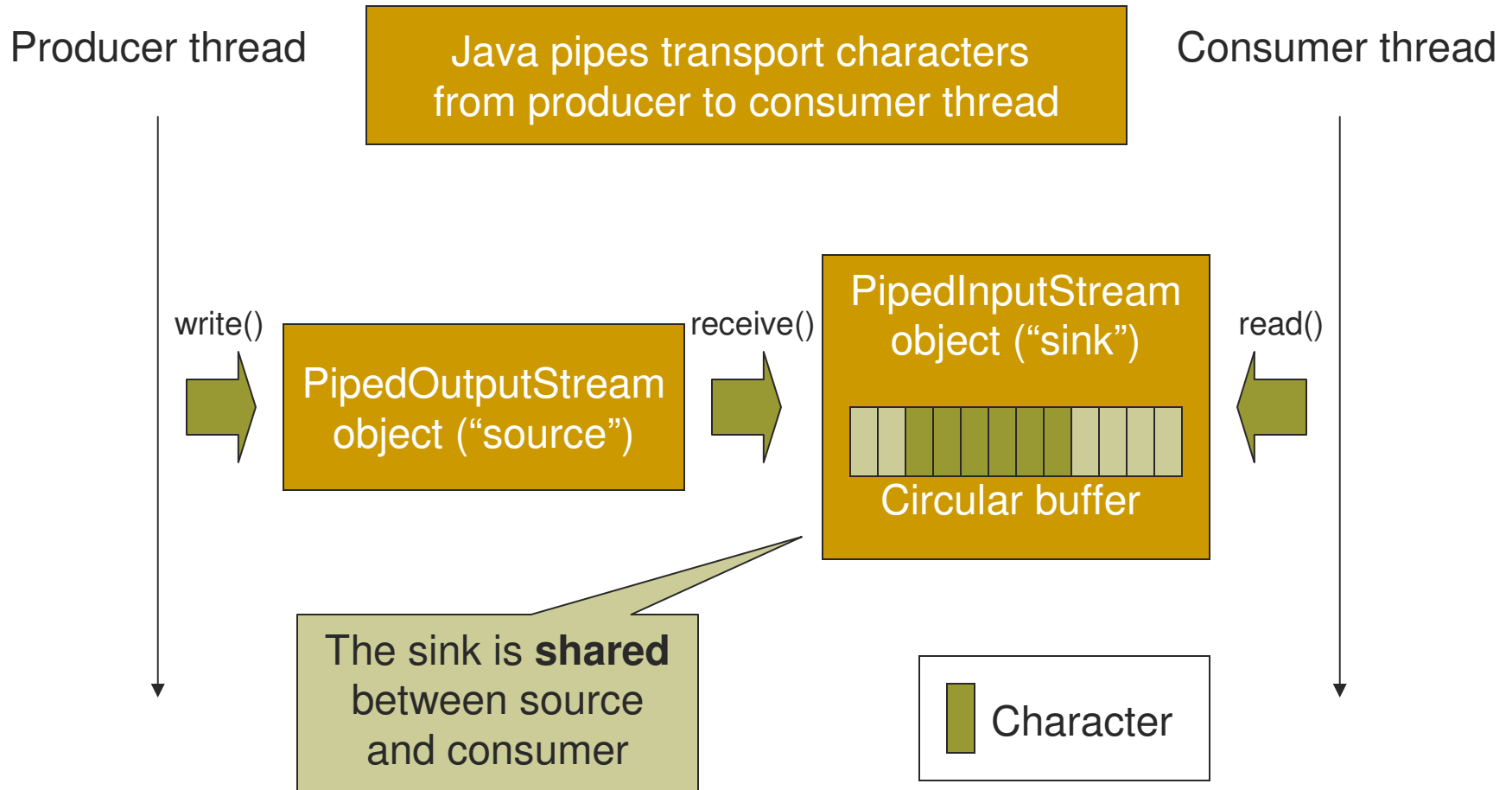
# Disjoint sets and capabilities can handle aliasing correctly

- Track disjoint sets of local aliases
  - Handle copies between local variables
- One capability for each object
  - Handle assignments to fields
- Capability annotations on methods
  - Handle aliasing during method calls

F. Smith, D. Walker & G. Morrisett. Alias types. In *European Symposium on Programming*, pages 366-381. Springer, 2000.

R. DeLine & M. Fähndrich. Enforcing high-level protocols in low-level software. In *ACM Conference on Programming Language Design and Implementation*, pages 59-69, 2001.

# Capabilities are not enough to specify or verify Java pipes

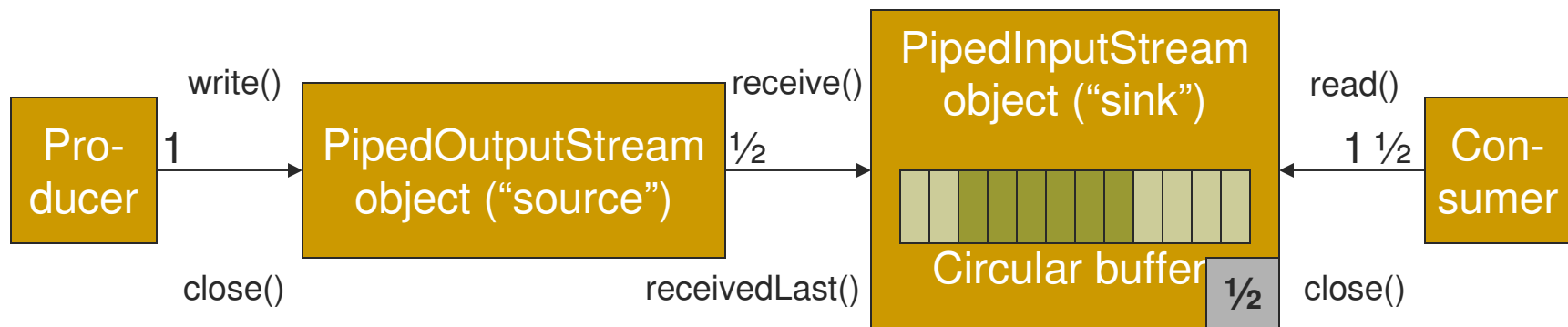


# Fractional permissions: Allow capabilities to be split and joined

- Permissions generalize capabilities
  - Permission required for all object access
  - Many permissions to the same object can exist
  - But keep track of how many permissions there are
- $1 \cdot x$  is the only permission for the referenced object
  - Similar to capability for  $x$
- $\frac{1}{2} \cdot x$  is one of two permissions for  $x$ 
  - $\frac{1}{2} \cdot x + \frac{1}{2} \cdot x = 1 \cdot x$

# Fractions for verifying that pipe is correctly closed

- Source and consumer each hold  $\frac{1}{2}$  fraction of sink
- Source uses its  $\frac{1}{2}$  to call `receive()` on sink
- Consumer uses its  $\frac{1}{2}$  to call `read()` on sink
- `ReceivedLast()` passes source's half to sink
- Consumer can call `close()` when sink in **eof** state
- `Close()` restores unique permission to sink and closes it



# Statically prevent error conditions

- Pipe implementation in Java standard library throws exceptions at runtime
  - If sink is closed before source
  - If source or sink are accessed after being closed
- Fractional permissions prevent these errors at compile time

J. Boyland. Checking interference with fractional permissions. In *Static Analysis Symposium*, LNCS vol. 2694, pages 55-72. Springer, 2003.  
K. Bierhoff & J. Aldrich. Modular typestate checking of aliased objects. In *OOPSLA'07*, pages 301-320, 2007.

# [ Agenda ]

---

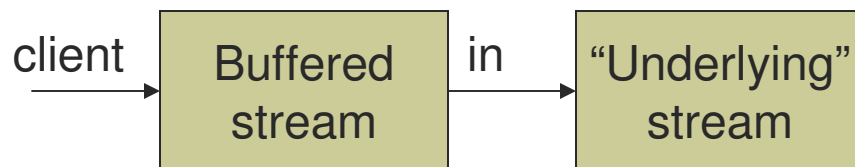
- Example protocols
    - Modeling protocols as state machines
  - Protocol analysis approaches
    - Annotations vs. interprocedural analyses
  - Aliasing challenges
    - Tracking aliases in methods and fields
- ➔ Protocol implementation checking ←

# Implementation checking tracks changes to fields

- So far we looked at clients
  - Code calling methods on sockets etc.
  - Assumed that declared protocol was right
- Checking protocol implementations
  - Does *this* change state as declared?
  - **State changes = field manipulations**
    - Protocols ensure that “something” happened already (or has not happened yet)
    - “Something” can (only) be recorded in fields



# State invariants define states in terms of fields

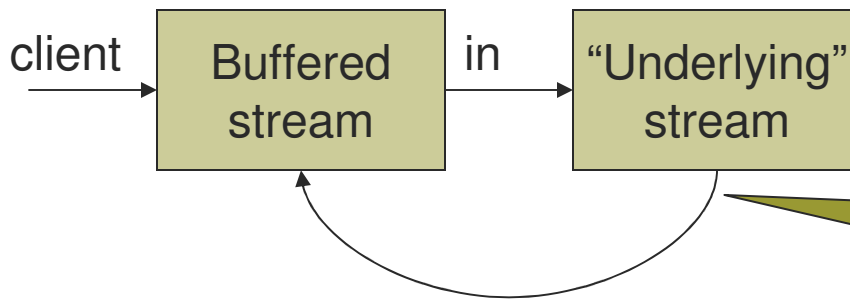


```
public class BufferedInputStream {  
    private InputStream in;  
    private byte[] buffer;  
    private int pos, count;  
    // open: in instate (within | eof) &&  
    //         buffer != null &&  
    //         0 ≤ pos ≤ count &&  
    //         count ≤ buffer.length  
    // closed: in == null && buffer == null
```

- **State invariants constrain fields...**
  - Constraints on field values
    - E.g., greater than zero or non-null
  - Expected state of referenced object
    - E.g., underlying stream should be “within” or “eof”
- ...but only while in a particular state

close() will change fields accordingly

# [ Don't forget aliasing...! ]



```
public class BufferedInputStream {  
    private InputStream in;  
    private byte[] buffer;  
    private int pos, count;  
    // open: in instate (within | eof) &&  
    //         buffer != null &&  
    //         0 ≤ pos ≤ count &&  
    //         count ≤ buffer.length  
    // closed: in == null && buffer == null
```

What happens when the underlying stream calls back to the buffer?

As it turns out, such a re-entrant callback can violate *count*'s invariant, leading to an access to *buffer* outside its bounds.

# [ Summary ]

---

- Protocols define temporal ordering of events
  - Can often be captured with state machines
- Protocol analysis needs to pay attention to
  - Interprocedural control flow
  - Aliasing of objects
- Disjoint sets and capabilities can handle aliasing correctly
  - Fractional permissions for heap sharing
- State changes correspond to field changes