Threads vs. Events

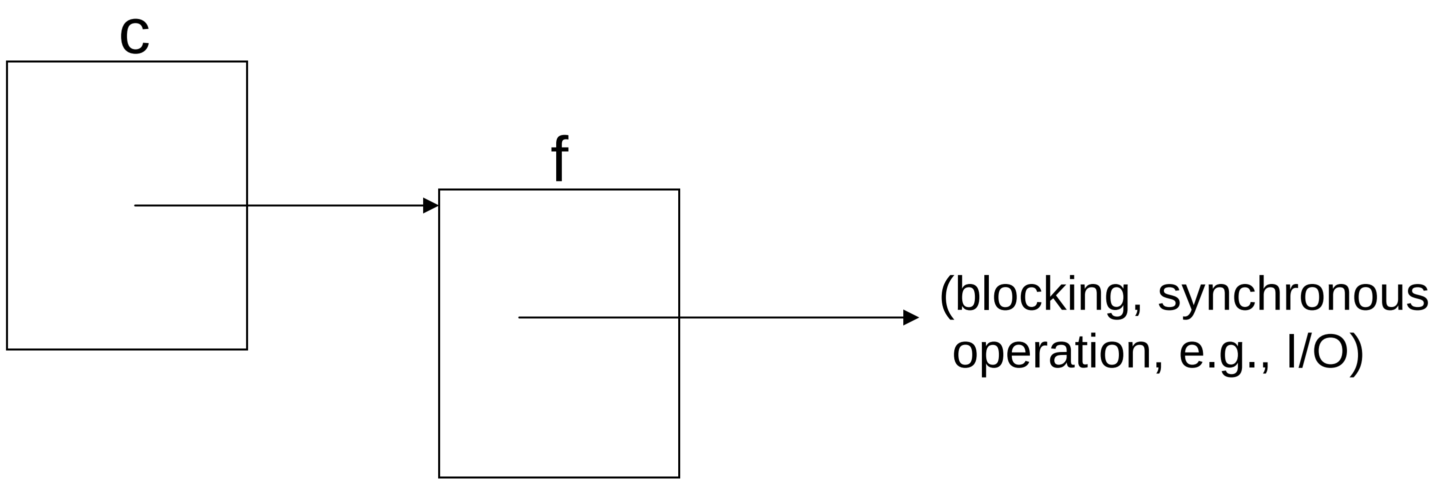
TAME – Event Style Programming with Threads

TAME

- expressive abstractions for event-based programming
- implemented via source-source translation
- avoids stack ripping
- type safety and composability via templates

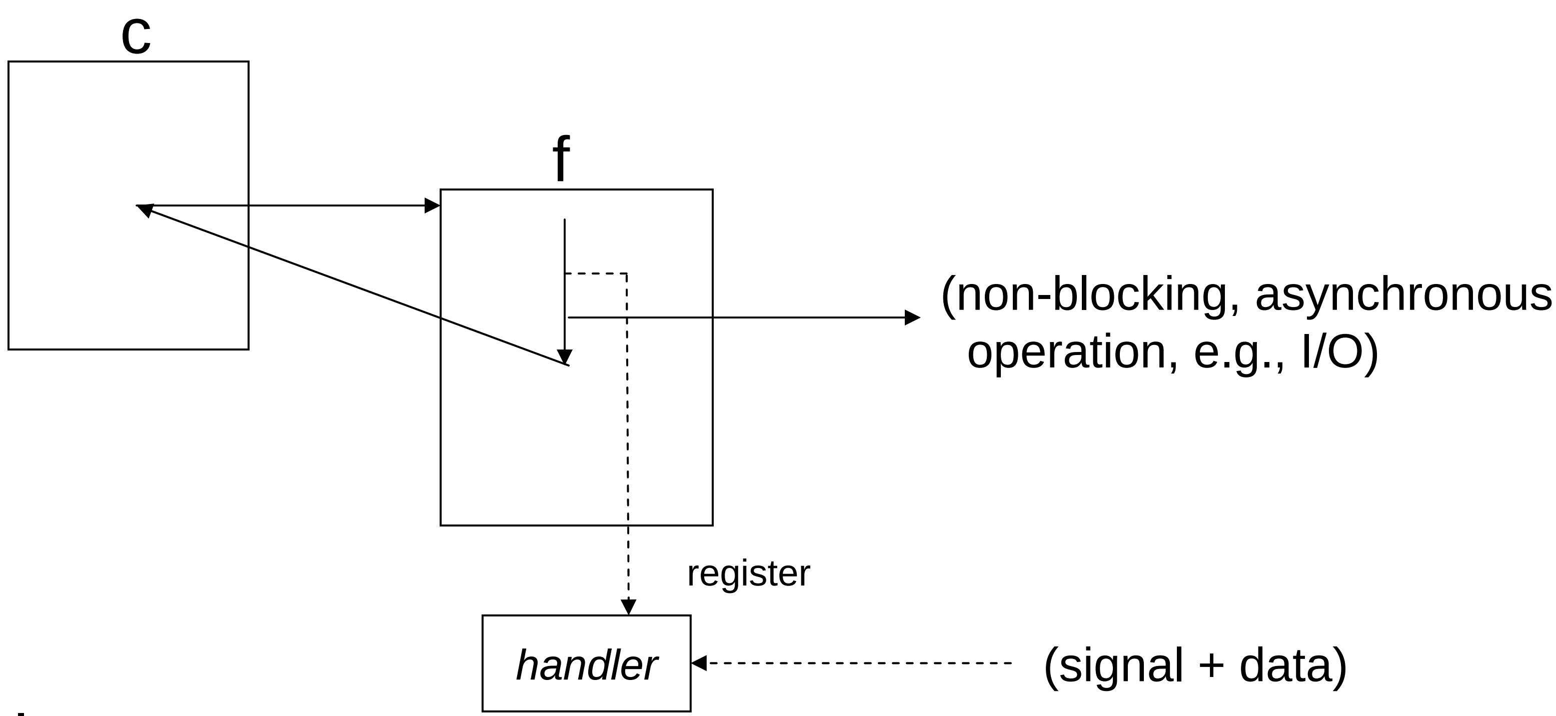
M. Krohn, E. Kohler, M.F. Kaashoek, "Events Can Make Sense," USENIX Annual Technical Conference, 2007, pp. 87-100.

A typical thread programming problem



Problem: the thread becomes blocked in the called routine (f) and the caller (c) is unable to continue even if it logically is able to do so.

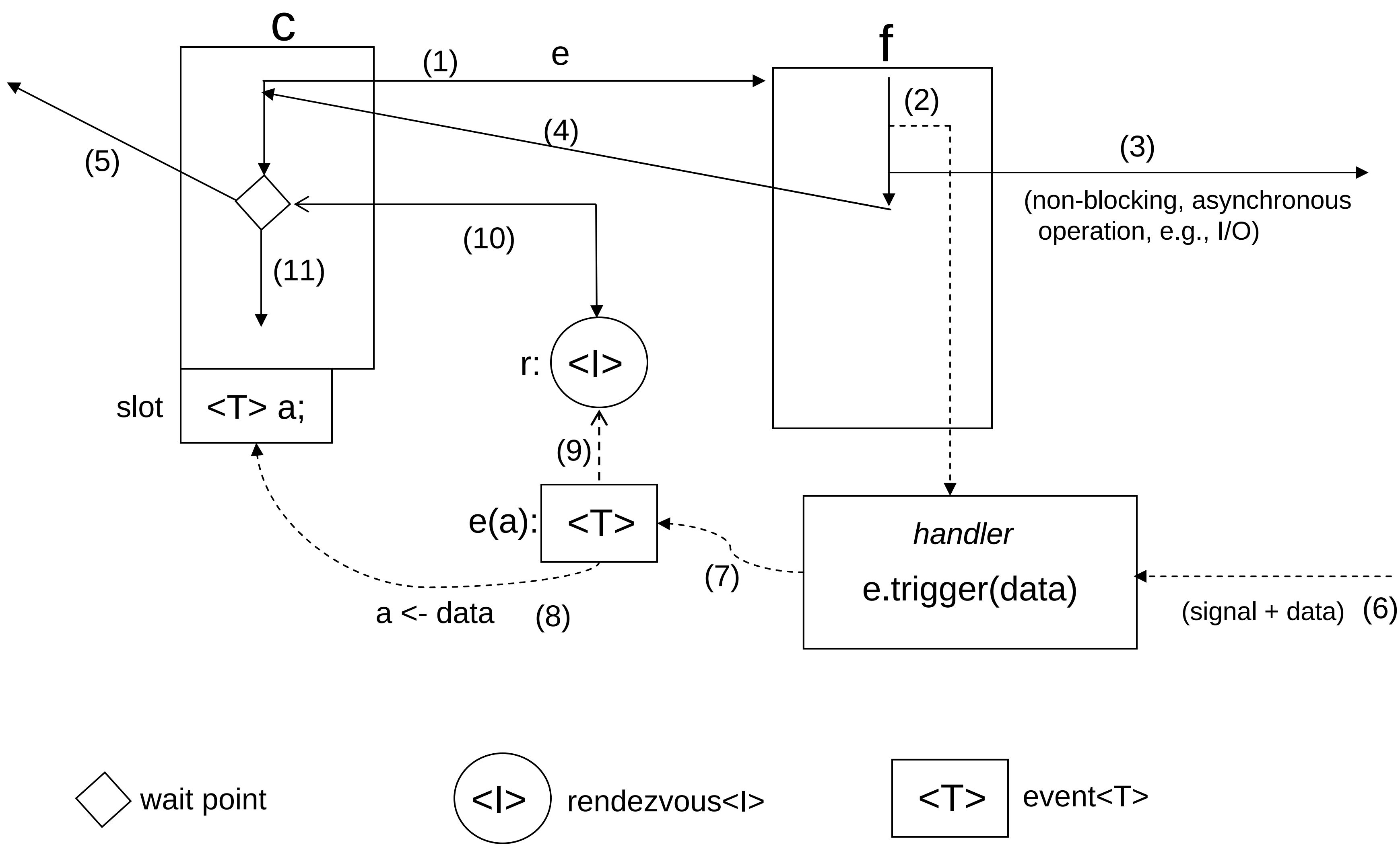
A partial solution



Issues

- Synchronization: how does the caller know when the signal has occurred without busy-waiting?
- Data: how does the caller know what data resulted from the operation?

A “Tame” solution



Tame Primitives

Classes	Keywords & Language Extensions	Functions & Methods

`event<>`
• A basic event.
`event<T>`
• An event with a single *trigger value* of type T. This value is set when the event occurs; an example might be a character read from a file descriptor. Events may also have multiple trigger values of types $T_1 \ldots T_n$.
`rendezvous<I>`
• Represents a set of outstanding events with event IDs of type I. Callers name a rendezvous when they block, and unblock on the triggering of any associated event.
	`twait(`r`[,`i`]);`
	• A wait point. Block on explicit rendezvous r, and optionally set the event ID i when control resumes.
	`tamed`
	• A return type for functions that use `twait`.
	`tvars { ... }`
	• Marks safe local variables.
	`twait {` *statements*`; }`
	• Wait point syntactic sugar: block on an implicit rendezvous until all events created in *statements* have triggered.
		`mkevent(`r`,`i`,`s`);`
		• Allocate a new event with event ID i. When triggered, it will awake rendezvous r and store trigger value in slot s.
		`mkevent(`s`);`
		• Allocate a new event for an implicit `twait{}` rendezvous. When triggered, store trigger value in slot s.
		e`.trigger(`v`);`
		• Trigger event e, with trigger value v.
		`timer(`to`,`e`); wait_on_fd(`fd`,`rw`,`e`);`
		• Primitive event interface for timeouts and file descriptor events, respectively.

Figure 2: Tame primitives for event programming in C++.

An example

```
void multidns(dnsname name[], ipaddr a[], int n) {
   for (int i = 0; i < n; i++)
      a[i] = gethostbyname(name[i]);
}
```

```
tamed multidns_tame(dnsname name[], ipaddr a[],
                                   int n, event<> done) {
   tvars { int i; }
   for (i = 0; i < n; i++)
      twait { gethost_ev(name[i], mkevent(a[i])); }
   done.trigger();
}
```

```
tamed gethost_ev(dsname name, event<ipaddr> e);
```

Variations on control flow

```
tamed multidns_par(dnsname name[], ipaddr a[],
                   int n, event<> done) {
   twait {
      for (int i = 0; i < sz; i++)
         gethost_ev(name[i], mkevent(a[i]));
   }
   done.trigger();
}
```

parallel control flow

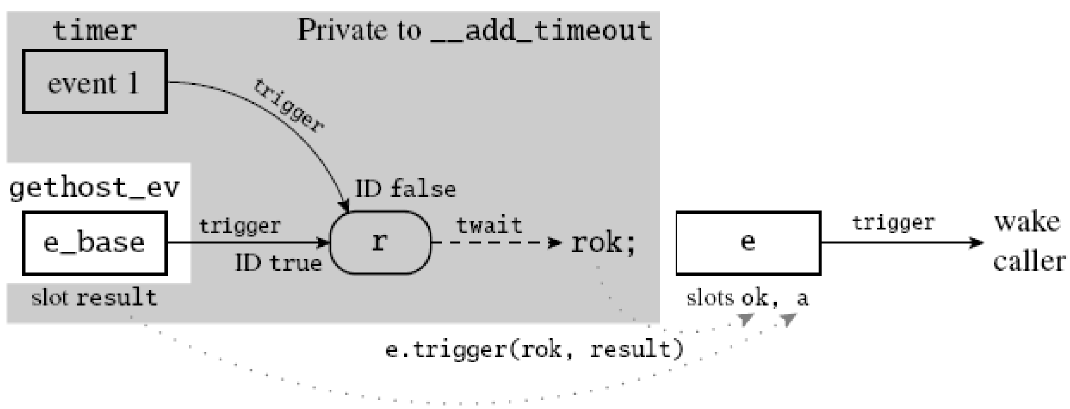
```
tamed multidns_win(dnsname name[], ipaddr a[],
                   int n, event<> done) {
  tvars { int sent(0), recv(0); rendezvous<> r; }
  while (recv < n)
     if (sent < n && sent - recv < WINDOWSIZE) {
        gethost_ev(name[sent], mkevent(r,a[sent]));
        sent++;
     } else {
        twait(r);
        recv++;
     }
  done.trigger();
}
```

window/pipeline control flow

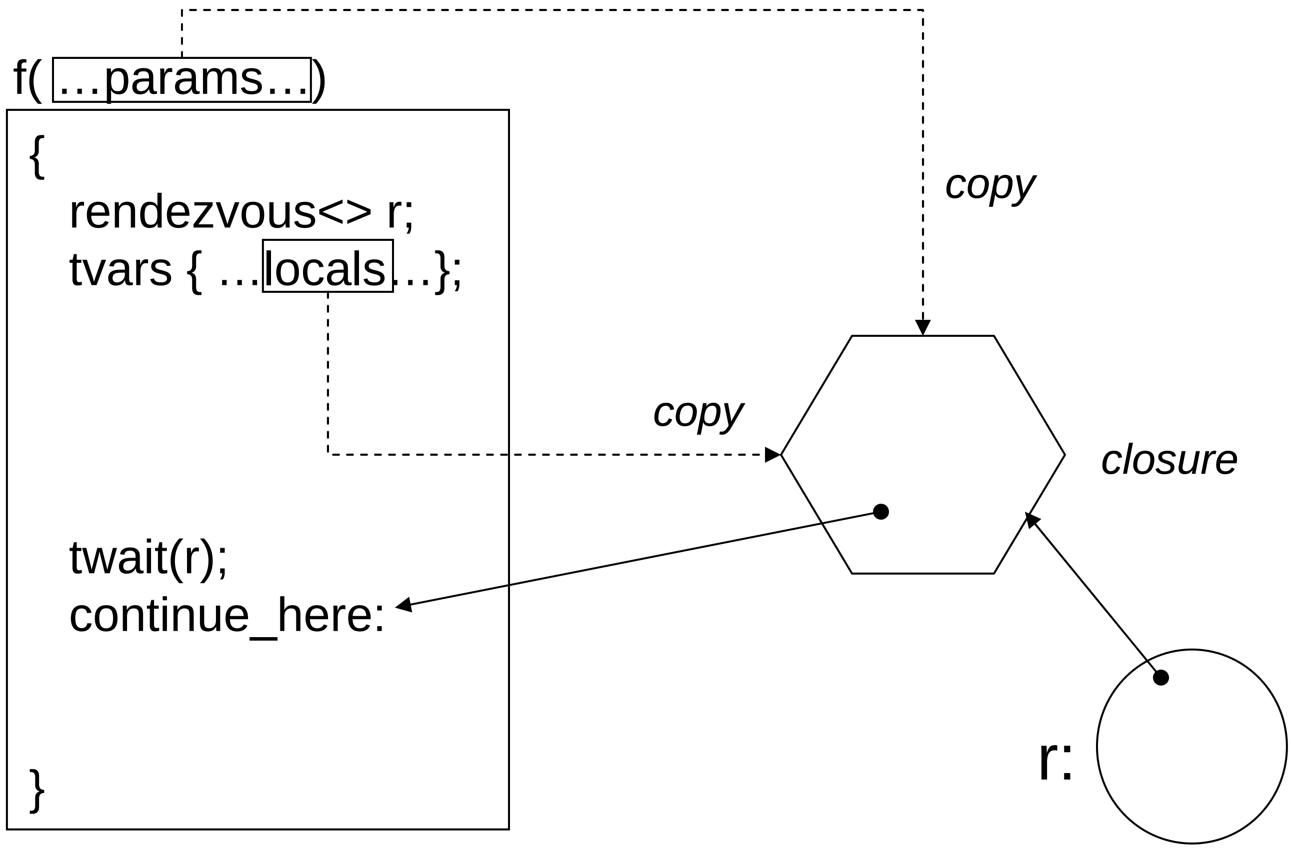
Event IDs & Composability

```
template <typename T> tamed
__add_timeout(event<T> &e_base, event<bool, T> e) {
    tvars { rendezvous<bool> r; T result; bool rok; }
    timer(TIMEOUT, mkevent(r, false));
    e_base = mkevent(r, true, result);
    twait(r, rok);
    e.trigger(rok, result);
    r.cancel();
}

template <typename T> event<T> add_timeout(event<bool, T> e) {
    event<T> e_base;
    __add_timeout(e_base, e);
    return e_base;
}
```



Closures



Smart pointers and reference counting insure correct deallocation of events, redezvous, and closures.

Performance
(relative to Capriccio)

	Capriccio	Tame
Throughput (connections/sec)	28,318	28,457
Number of threads	350	1
Physical memory (kB)	6,560	2,156
Virtual memory (kB)	49,517	10,740

Figure 7: Measurements of Knot at maximum throughput. Throughput is averaged over the whole one-minute run. Memory readings are taken after the warm-up period, as reported by ps.