Nabil S. Al Ramli

TAME

Event Based Concurrency System

Virginia Tech

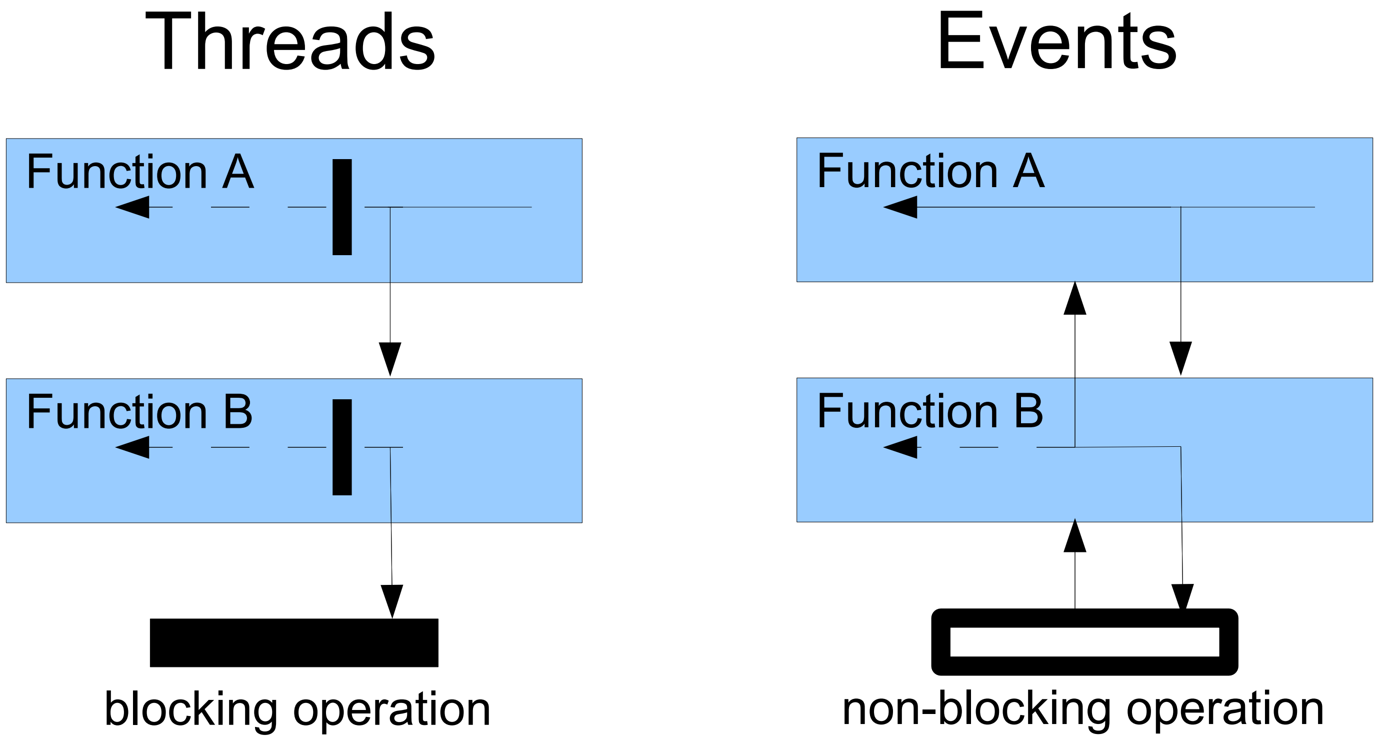
What is TAME?

- Event based concurrency system
    - **C++ libraries and source-to-source translation**
    - **For network applications**
    - **No “stack-ripping”**
    - **Type safe (templates)**
    - **Performance of events**
    - **Programmabiliy of threads**

Events Vs. Threads

Measure	Threads	Events
Support for extremely high concurrency	X	√
Portability	X	√
Better performance, Less use of memory	X	√
No stack-ripping	X	√

Events Vs. Threads



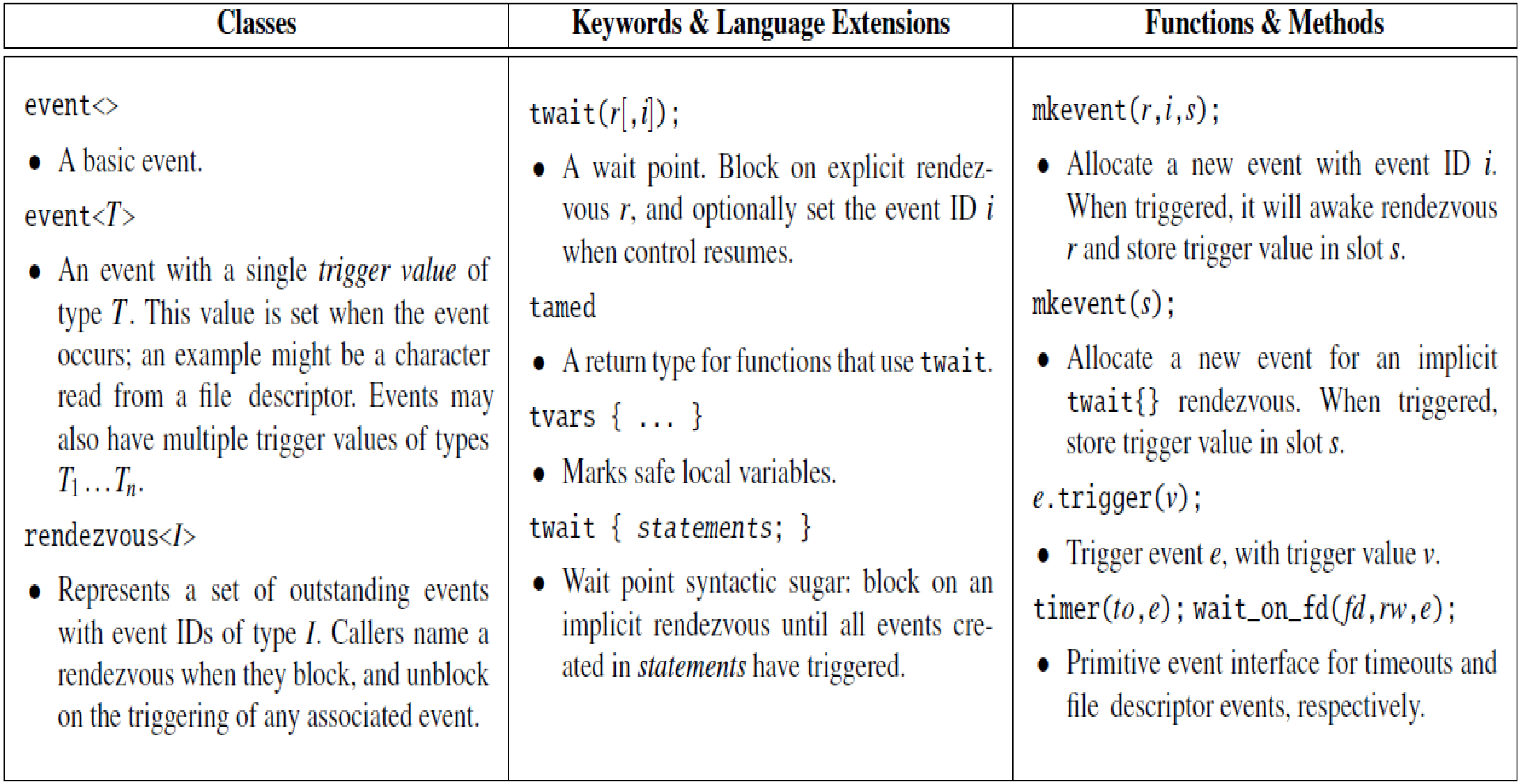
Semantics Example

```
// Threads
void wait_then_print_threads() {
sleep(10); // blocks this function and all callers
printf("Done!");
}
```

```
// Tame primitives
tamed wait_then_print_tame() {
tvars { rendezvous<> r; }
event<> e = mkevent(r); // allocate event on r
timer(10, e); // cause e to be triggered after 10 sec
twait(r); // block until an event on r is triggered
// only blocks this function, not its callers!
printf("Done!");
}
```

```
// Tame syntactic sugar
tamed wait_then_print_simple_tame() {
twait { timer(10, mkevent()); }
printf("Done!");
}
```

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++.

Wait Points

- blocks calling function until 1+ events triggered
- function returns to its caller
- but the function does not complete
- execution point and local vars are preserved
- When an event occurs, the function “unblocks” and resumes processing at the wait point

```
twait { statements; }
```

```
{ rendezvous<> __r;
statements; // where mkevent calls create events on __r
while (not all __r events have completed)
twait(__r); }
```

Safe Local Variables

- Their values are preserved across wait points
- Preserved in a heap-allocated closure
- `tvars {}`

- Unsafe local variables have indeterminate values after a wait point
- Compile warnings when a local variable should be made safe

Closures

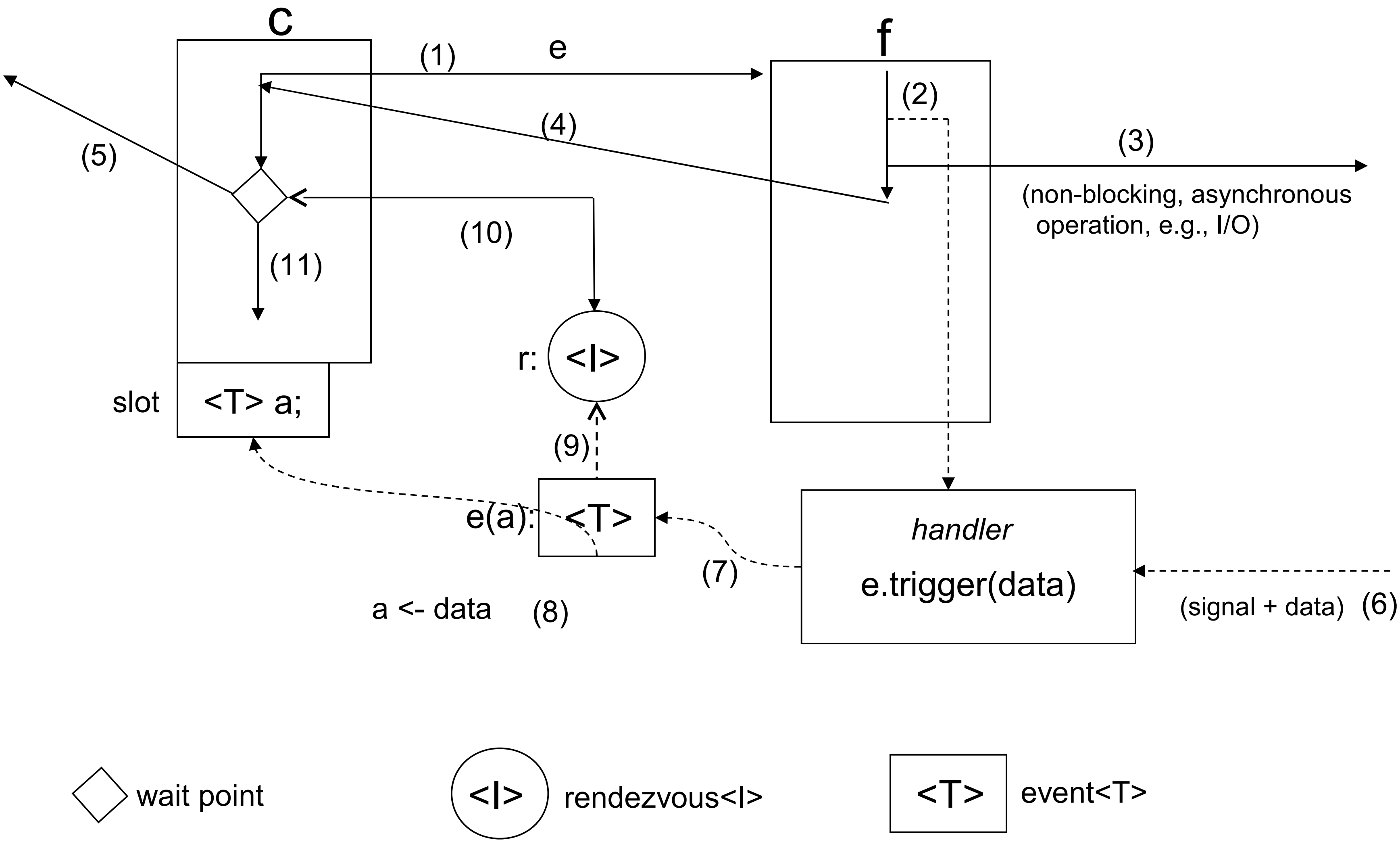
- Internally, Tame writes a new C++ structure for each tamed function
- Each `tamed` function has one closure, contains:
  - **Next statement to execute**
  - **Function parameters**
  - `tvars` **variables**
  - `rendezvouz`

Closures Translation

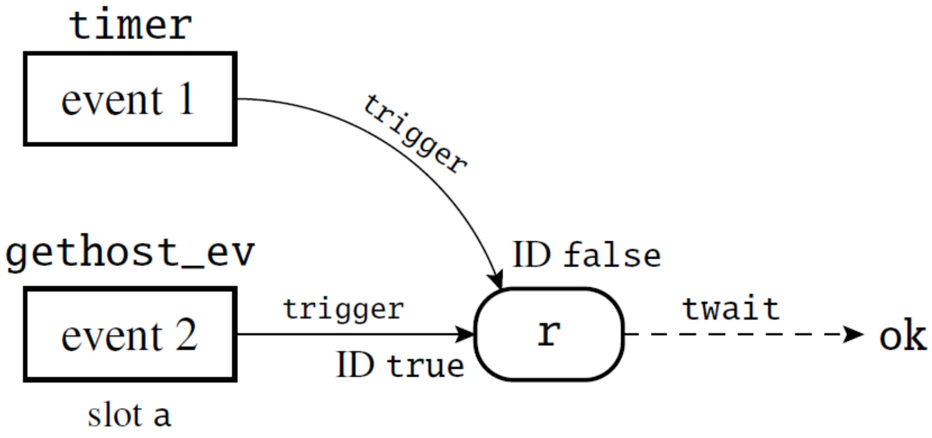
```
tamed A::f(int x) {
tvars { rendezvous<> r; }
a(mkevent(r)); twait(r); b(); }
```

```
void A::f(int __tame_x, A_f_closure *__cls) {
if (__cls == 0)
__cls = new A__f__closure(this, &A::fn, __tame_x);
assert(this == __cls->this_A);
int &x = __cls->x;
rendezvous<> &r = __cls->r;
switch (__cls->entry_point) {
case 0: // original entry
goto __A__f__entry__0;
case 1: // reentry after first twait
goto __A__f__entry__1; }
__A__f__entry__0:
a(_mkevent(__cls,r));
if (!r.has_queued_trigger()) {
__cls->entry_point = 1;
r.set_reenter_closure(__cls);
return; }
__A__f__entry__1:
b();
}
```

A “Tame” solution



Composeability



timer

event 1

trigger

gethost_ev

event 2

slot a

trigger

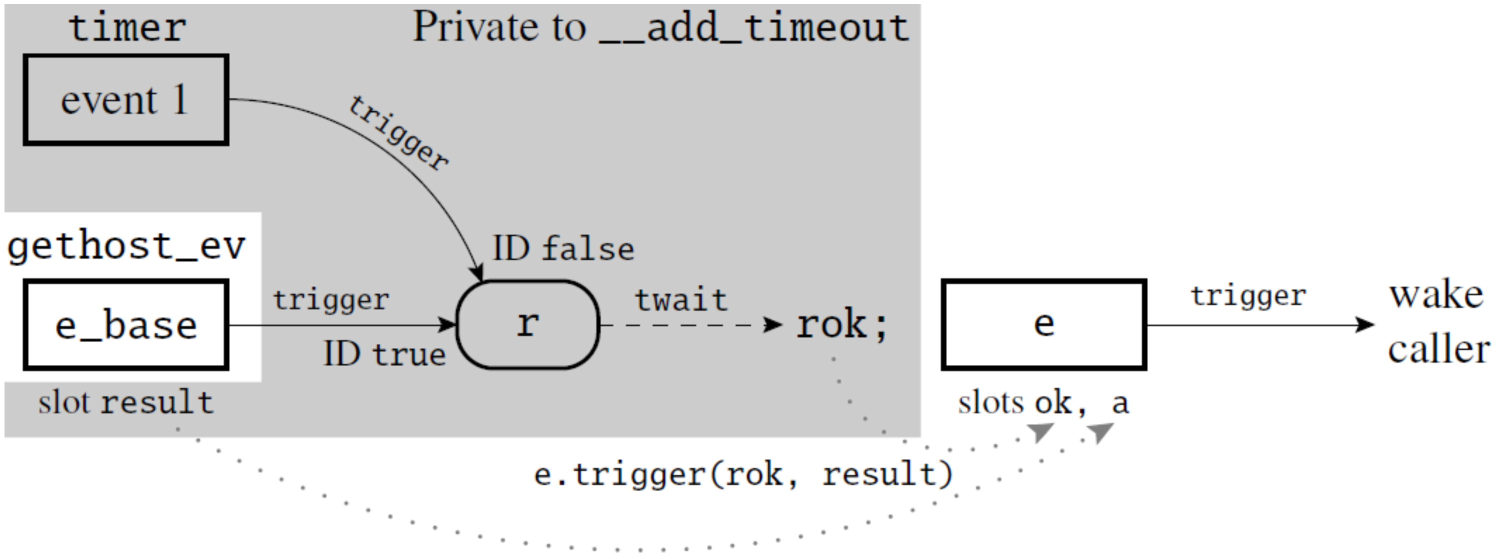
ID true

ID false

r

twait

ok



timer

Private to __add_timeout

event 1

trigger

gethost_ev

e_base

slot result

trigger

ID true

ID false

r

twait

rok;

e

slots ok, a

trigger

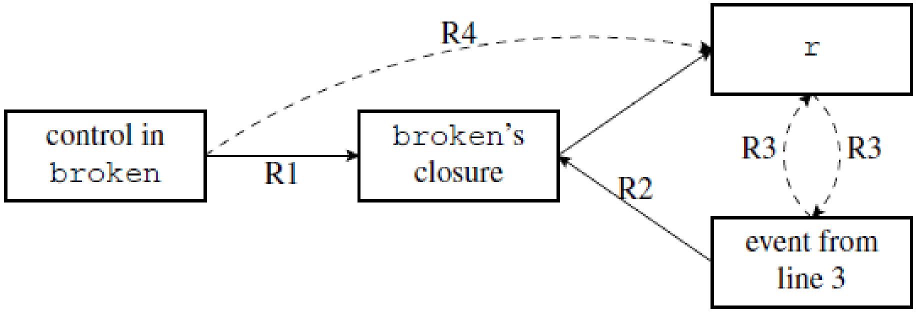
wake caller

e.trigger(rok, result)

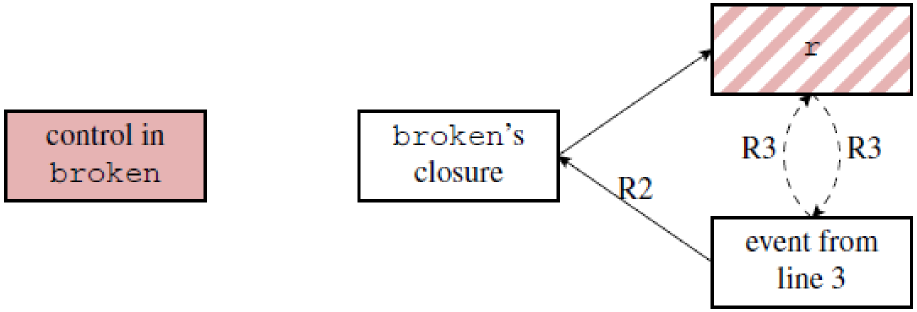
Memory Management

```
tamed broken(dnsname nm) {
  tvars { rendezvous<> r; ipaddr res; }
  gethost_ev(nm, mkevent(r, res));
  // Whoops: forgot to twait(r)!
}
```

- Solution is reference-counting
  - **Keep track of events and closures**
  - **C++ “smart pointer” classes**
- Two types of reference counts
  - **Strong references**
    - are conventional reference counts
  - **Weak references**
    - Allow access to the object only if not deallocated



(a) After the event allocation on line 3.



(b) After control exits broken on line 5.

Related Work

System	Similarities	Differences
Capriccio	•Uses events •Automatic stack management	•Thread interface
Adya et al.	•Uses events	•Some manual stack management
SEDA	•Can be used together with TAME	•Uses threads and events