P^3: A Profiler Suite for Parallel Applications on the Java Virtual Machine

Andrea Rosà, Walter Binder

Università della Svizzera italiana, Switzerland

Università della Svizzera italiana

APLAS 2020
November 30, 2020

P^3

- Novel **profiling suite** for parallel applications running on the Java Virtual Machine (JVM)
- Focus on metrics related to parallelism, concurrency, and synchronization
- Novel metrics:
  - Volatile memory accesses
  - Futures and promises
  - Synchronizers
  - Synchronized collections
  - Concurrent collections

P^3

- ➢ Challenges in developing P^3:
  - Moderate overhead
  - High accuracy
- ➢ Enabling features:
  - Use of lock-free data structures
  - Few computations done in instrumentation
  - Use of reification of reflective information in a separate instrumentation process [1]

[1] A. Rosà et al., "Optimizing Type-specific Instrumentation on the JVM with Reflective Supertype Information". Journal of Visual Languages & Computing 49, 2018.

Metrics

Module	**Metrics**
thread	Threads start and termination.
task	Tasks creation and execution (instances of Runnable, Callable and ForkJoinTask).
actor	Use of Akka actors.
future	Futures and promises from Java's, Scala's and Twitter's libraries.
ilock	Implicit locks: use of synchronized methods and blocks.
elock	Explicit locks: use of interfaces Lock, ReadWriteLock and Condition.
wait	Calls to Object.wait, Object.notify and Object.notifyAll.
join	Calls to Thread.join.
park	Thread parking and unparking.
synch	Synchronizers: Semaphore, CountDownLatch, CyclicBarrier, Phaser and Exchanger.
cas	Compare-and-swap (CAS), get-and-swap (GAS), get-and-add (GAA).
atomic	Use of atomic classes: AtomicInt, AtomicLong, AtomicReference.
volatile	Accesses to volatile fields.
scoll	Use of synchronized collections.
ccoll	Use of concurrent collections: BlockingQueue, ConcurrentMap and subtypes.

Metrics

Module	Metrics
thread	Threads start and termination.
task	Tasks creation and execution (instances of Runnable, Callable and ForkJoinTask).
actor	Use of Akka actors.
future	Futures and promises from Java's, Scala's and Twitter's libraries.
ilock	Implicit locks: use of synchronized methods and blocks.
elock	Explicit locks: use of interfaces Lock, ReadWriteLock and Condition.
wait	Calls to Object.wait, Object.notify and Object.notifyAll.
join	Calls to Thread.join.
park	Thread parking and unparking.
synch	Synchronizers: Semaphore, CountDownLatch, CyclicBarrier, Phaser and Exchanger.
cas	Compare-and-swap (CAS), get-and-swap (GAS), get-and-add (GAA).
atomic	Use of atomic classes: AtomicInt, AtomicLong, AtomicReference.
volatile	Accesses to volatile fields.
scoll	Use of synchronized collections.
ccoll	Use of concurrent collections: BlockingQueue, ConcurrentMap and subtypes.

Metrics

Module	Metrics
thread	Threads start and termination.
task	Tasks creation and execution (instances of Runnable, Callable and ForkJoinTask).
actor	Use of Akka actors.
future	Futures and promises from Java's, Scala's and Twitter's libraries.
ilock	Implicit locks: use of synchronized methods and blocks.
elock	Explicit locks: use of interfaces Lock, ReadWriteLock and Condition.
wait	Calls to Object.wait, Object.notify and Object.notifyAll.
join	Calls to Thread.join.
park	Thread parking and unparking.
synch	Synchronizers: Semaphore, CountDownLatch, CyclicBarrier, Phaser and Exchanger.
cas	Compare-and-swap (CAS), get-and-swap (GAS), get-and-add (GAA).
atomic	Use of atomic classes: AtomicInt, AtomicLong, AtomicReference.
volatile	Accesses to volatile fields.
scoll	Use of synchronized collections.
ccoll	Use of concurrent collections: BlockingQueue, ConcurrentMap and subtypes.

Metrics

Module	Metrics
thread	Threads start and termination.
task	Tasks creation and execution (instances of Runnable, Callable and ForkJoinTask).
actor	Use of Akka actors.
future	Futures and promises from Java's, Scala's and Twitter's libraries.
ilock	Implicit locks: use of synchronized methods and blocks.
elock	Explicit locks: use of interfaces Lock, ReadWriteLock and Condition.
wait	Calls to Object.wait, Object.notify and Object.notifyAll.
join	Calls to Thread.join.
park	Thread parking and unparking.
synch	Synchronizers: Semaphore, CountDownLatch, CyclicBarrier, Phaser and Exchanger.
cas	Compare-and-swap (CAS), get-and-swap (GAS), get-and-add (GAA).
atomic	Use of atomic classes: AtomicInt, AtomicLong, AtomicReference.
volatile	Accesses to volatile fields.
scoll	Use of synchronized collections.
ccoll	Use of concurrent collections: BlockingQueue, ConcurrentMap and subtypes.

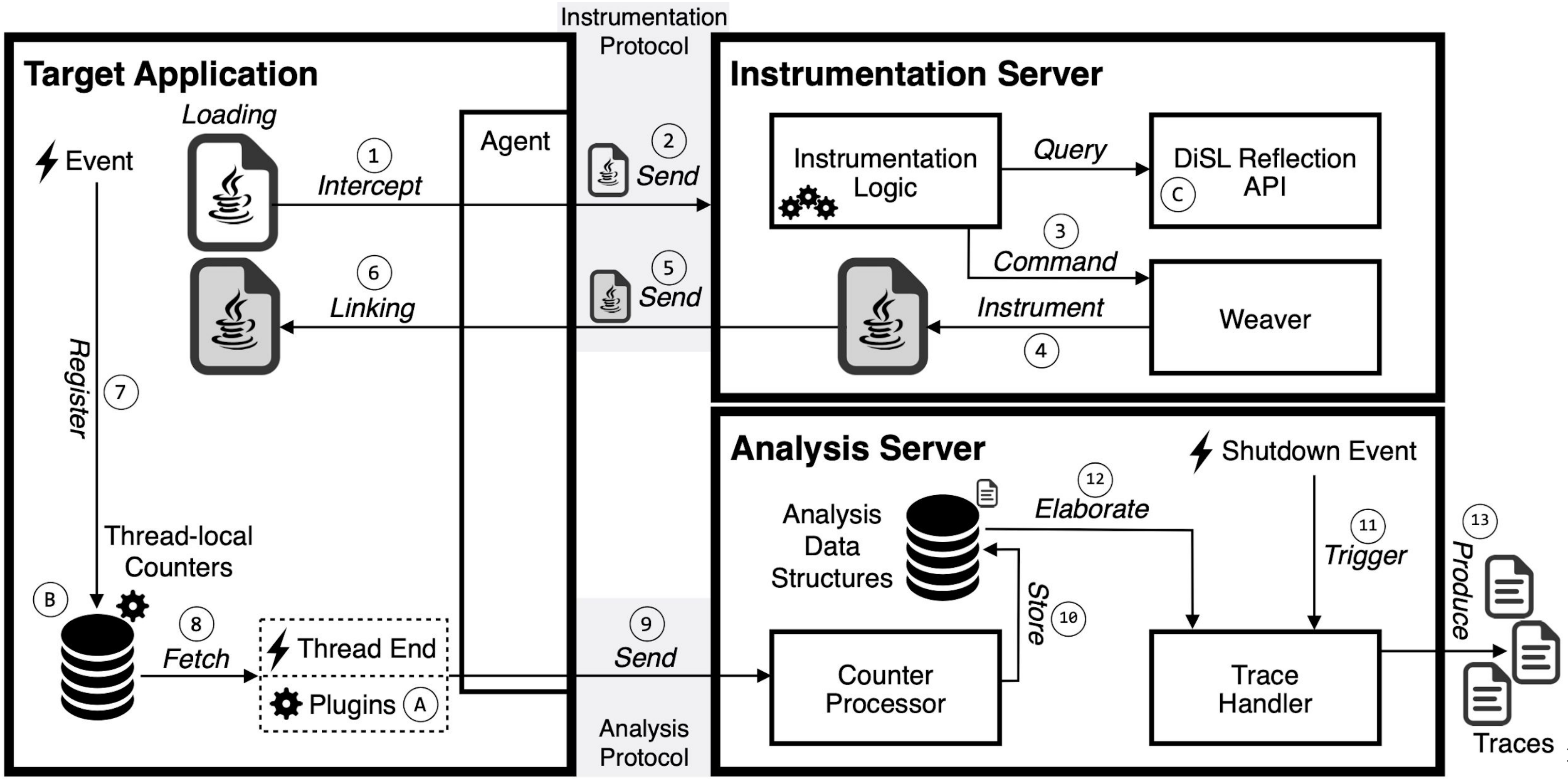
Metrics

Module	Metrics
thread	Threads start and termination.
task	Tasks creation and execution (instances of Runnable, Callable and ForkJoinTask).
actor	Use of Akka actors.
future	Futures and promises from Java's, Scala's and Twitter's libraries.
ilock	Implicit locks: use of synchronized methods and blocks.
elock	Explicit locks: use of interfaces Lock, ReadWriteLock and Condition.
wait	Calls to Object.wait, Object.notify and Object.notifyAll.
join	Calls to Thread.join.
park	Thread parking and unparking.
synch	Synchronizers: Semaphore, CountDownLatch, CyclicBarrier, Phaser and Exchanger.
cas	Compare-and-swap (CAS), get-and-swap (GAS), get-and-add (GAA).
atomic	Use of atomic classes: AtomicInt, AtomicLong, AtomicReference.
volatile	Accesses to volatile fields.
scoll	Use of synchronized collections.
ccoll	Use of concurrent collections: BlockingQueue, ConcurrentMap and subtypes.

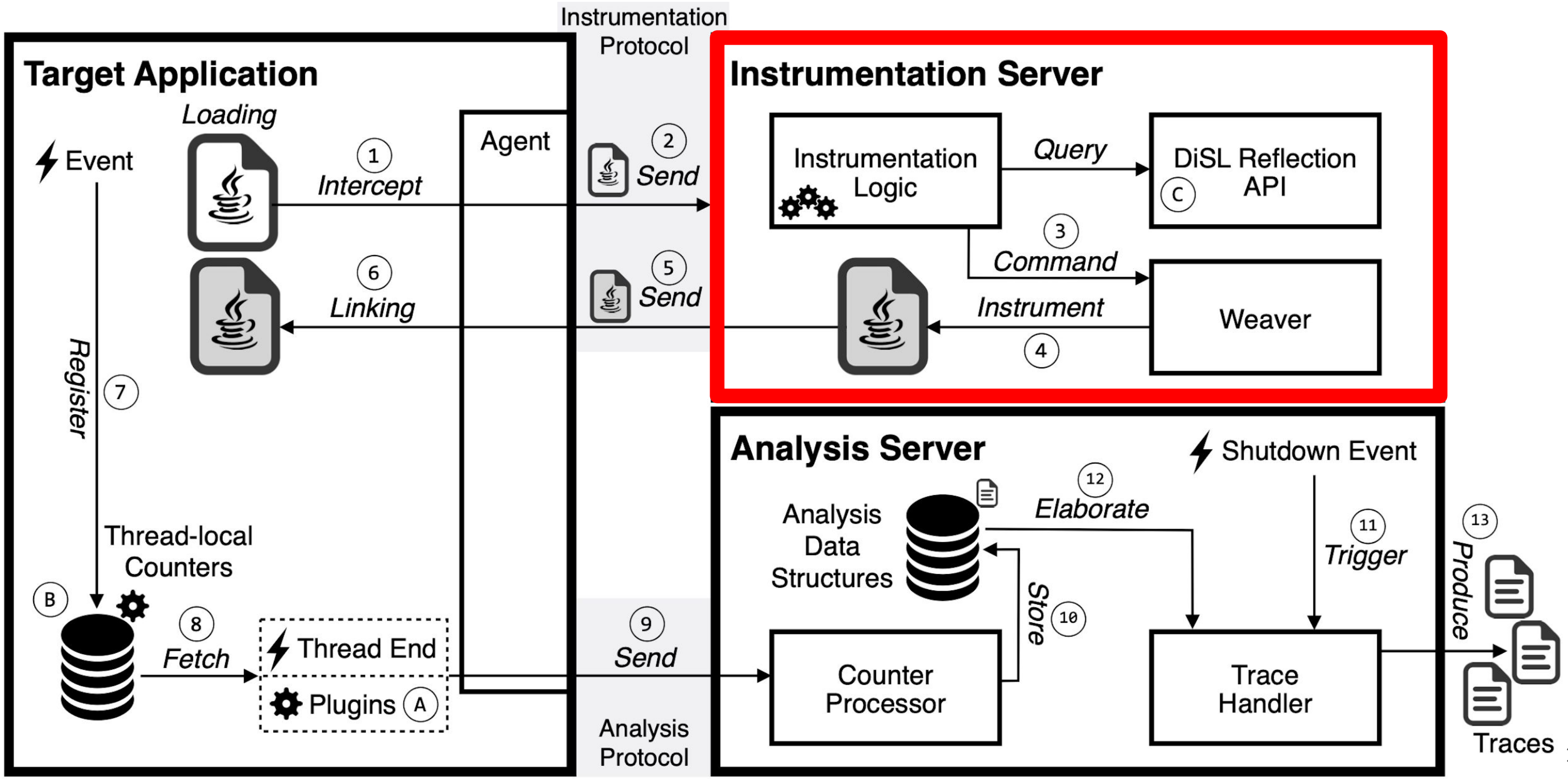
Additional Metrics

- ➢ Bytecode count
  - ● Allows metric normalization
  - ● Useful for comparing metrics in different applications
- ➢ Caller context
  - ● Allows per-method event counters
  - ● Enable detection of code where most events of a given type occur

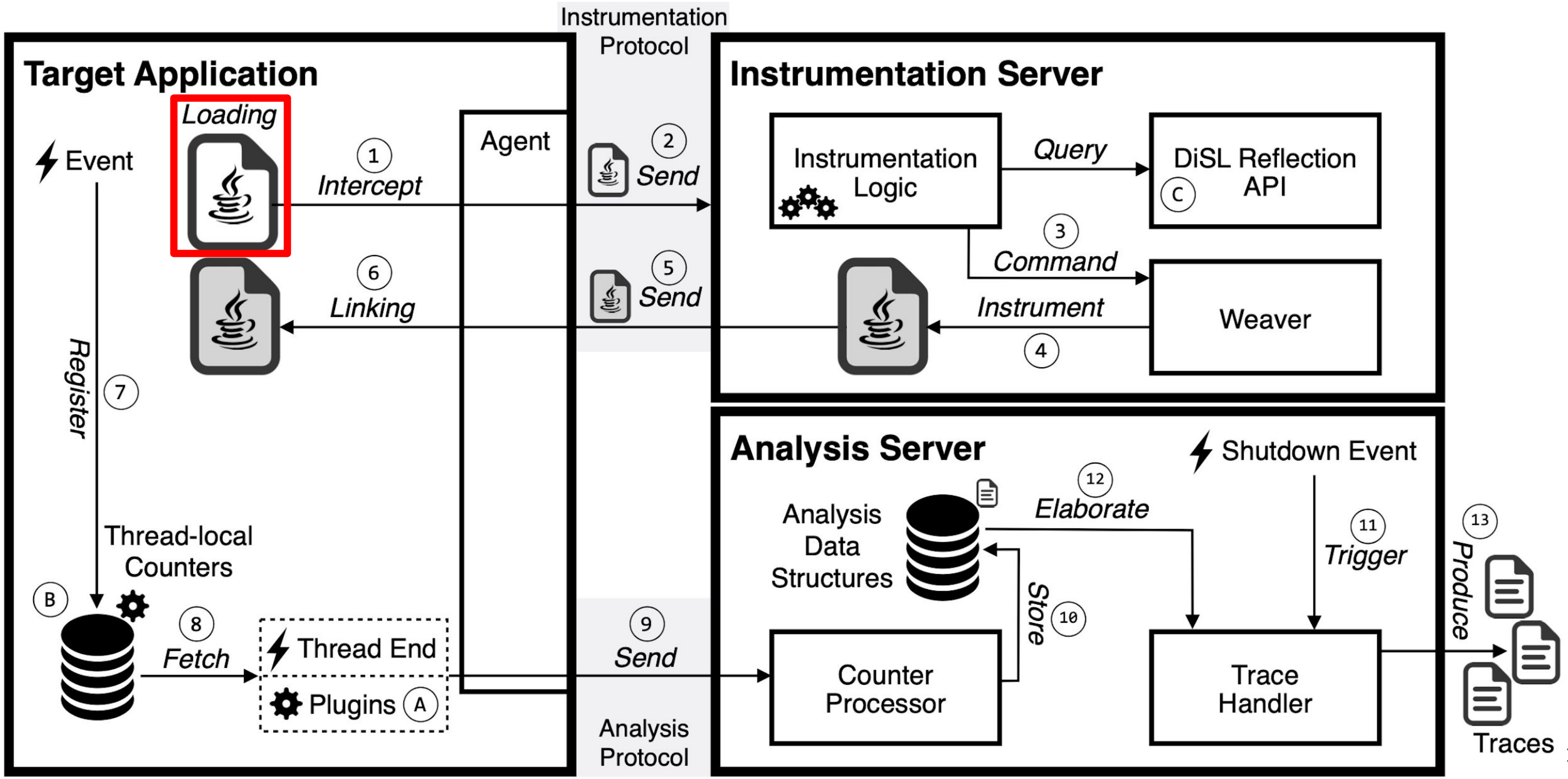
Architecture



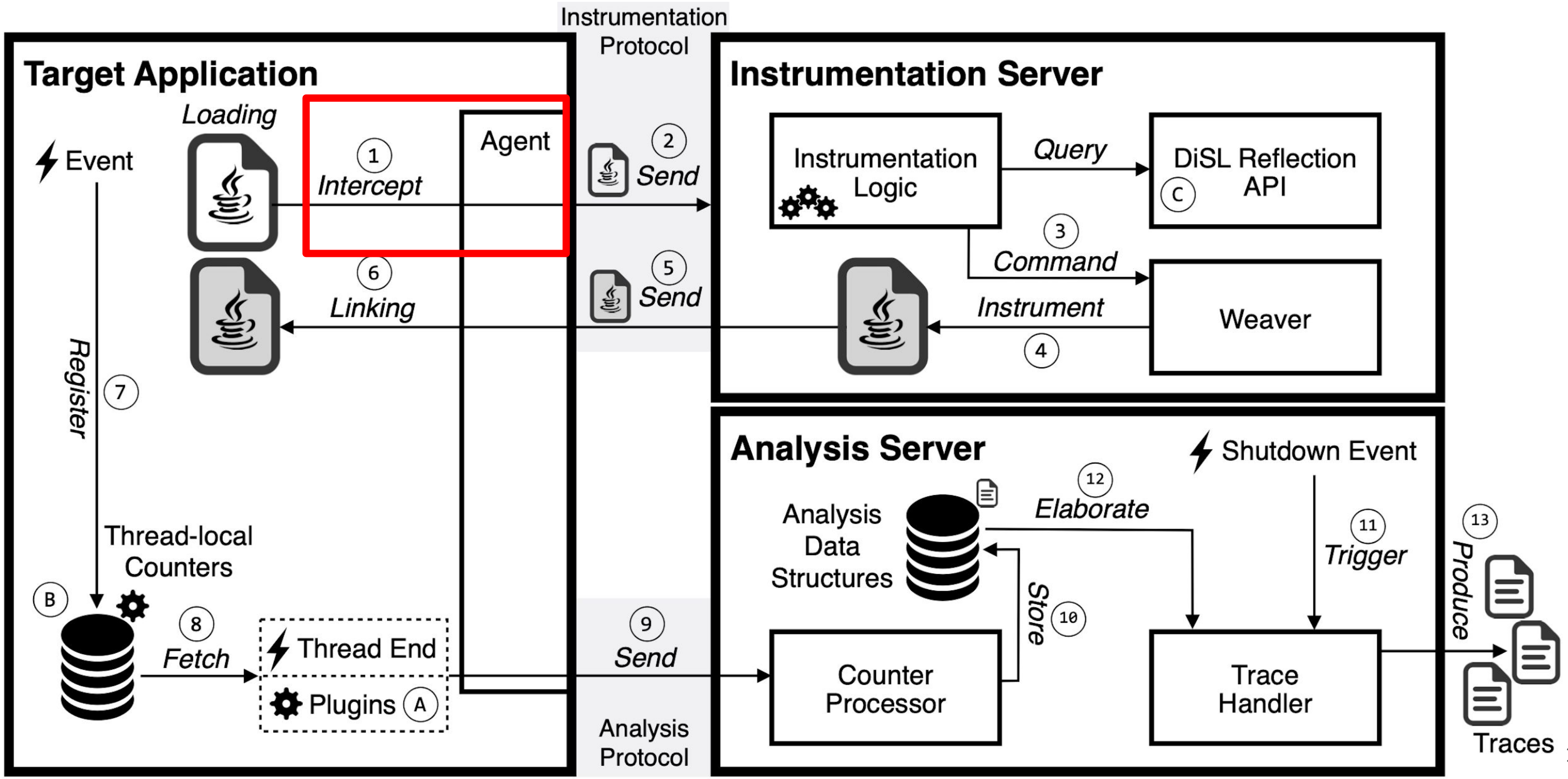
Architecture



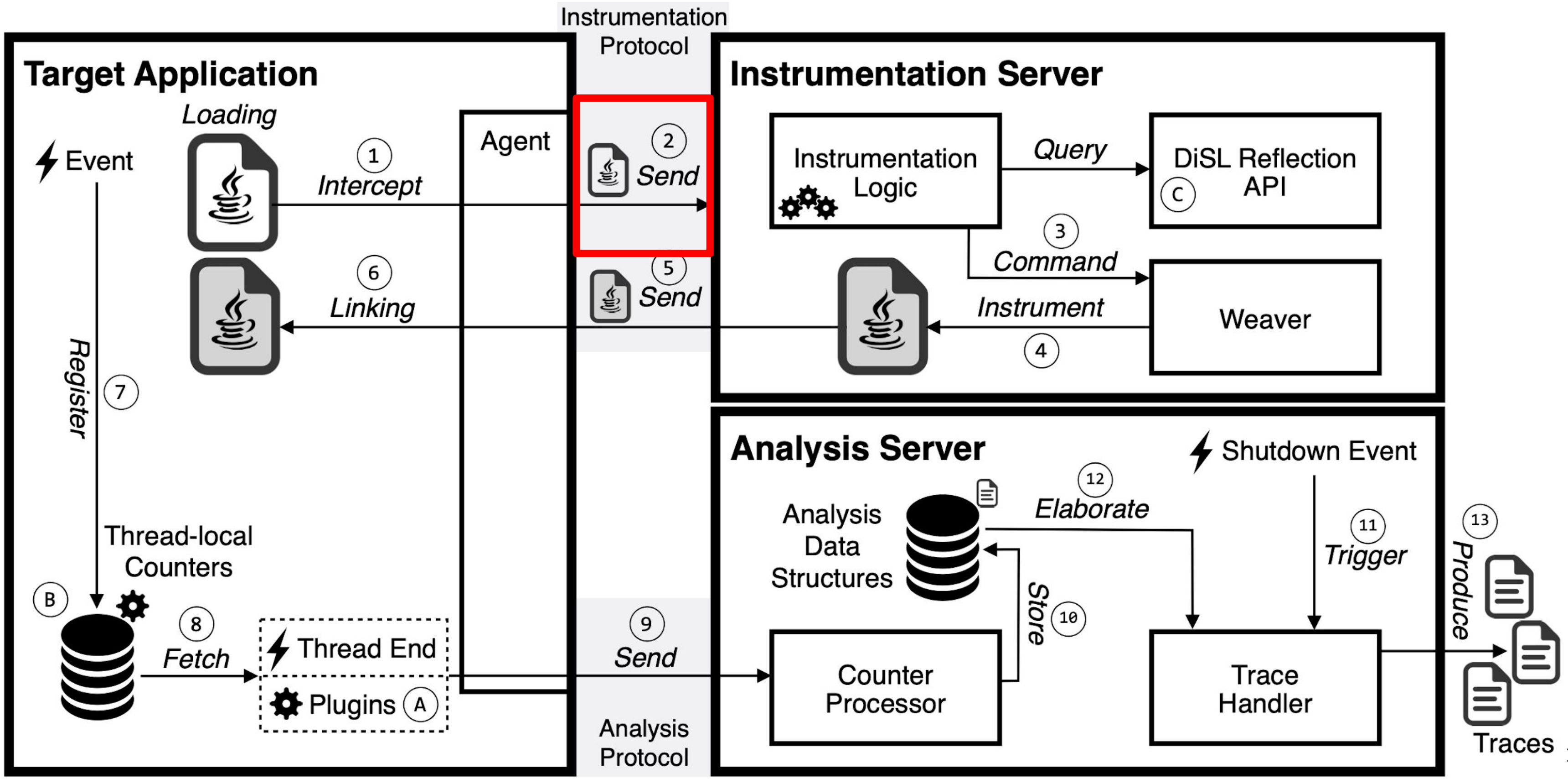
Architecture



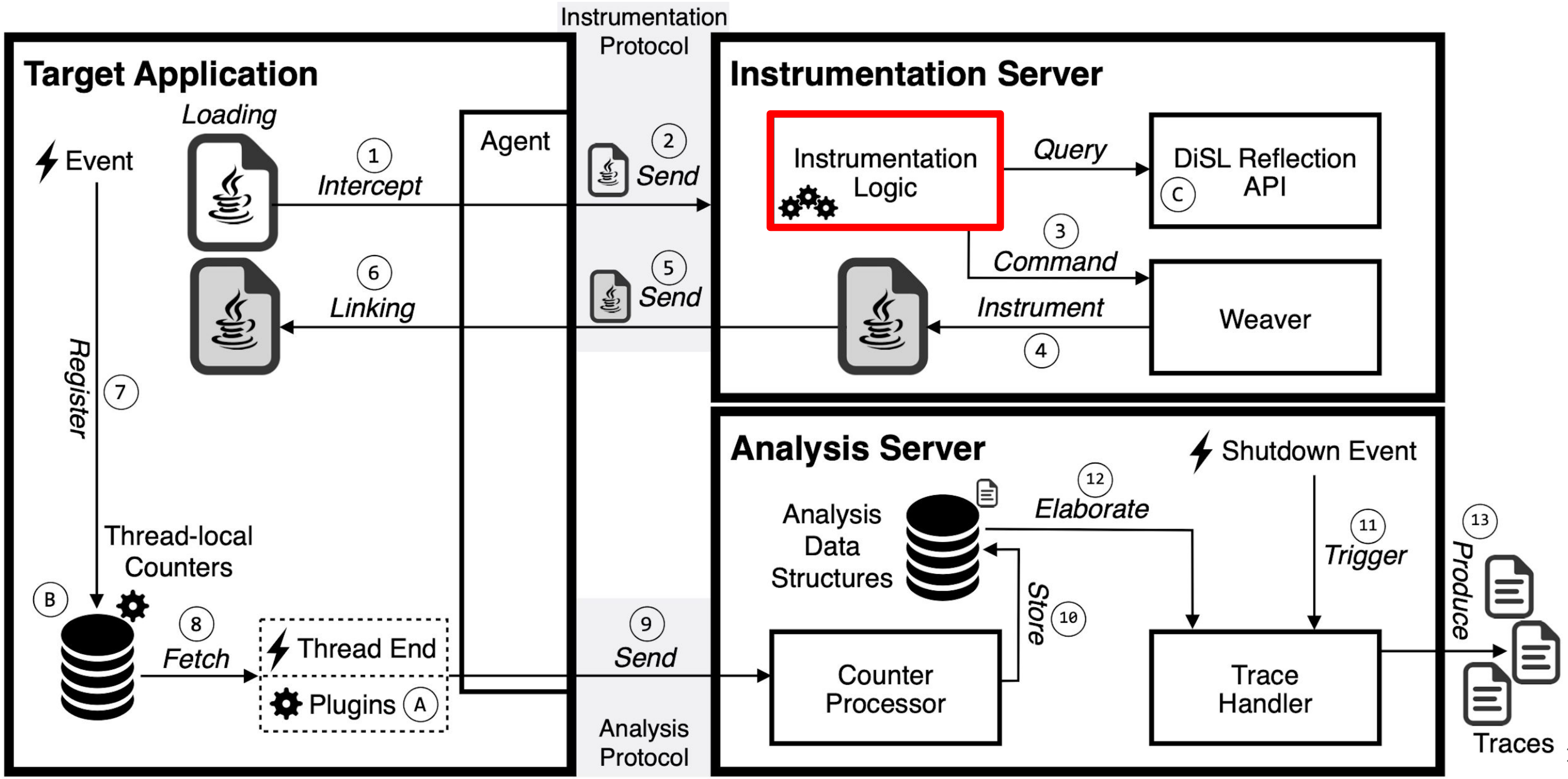
Architecture



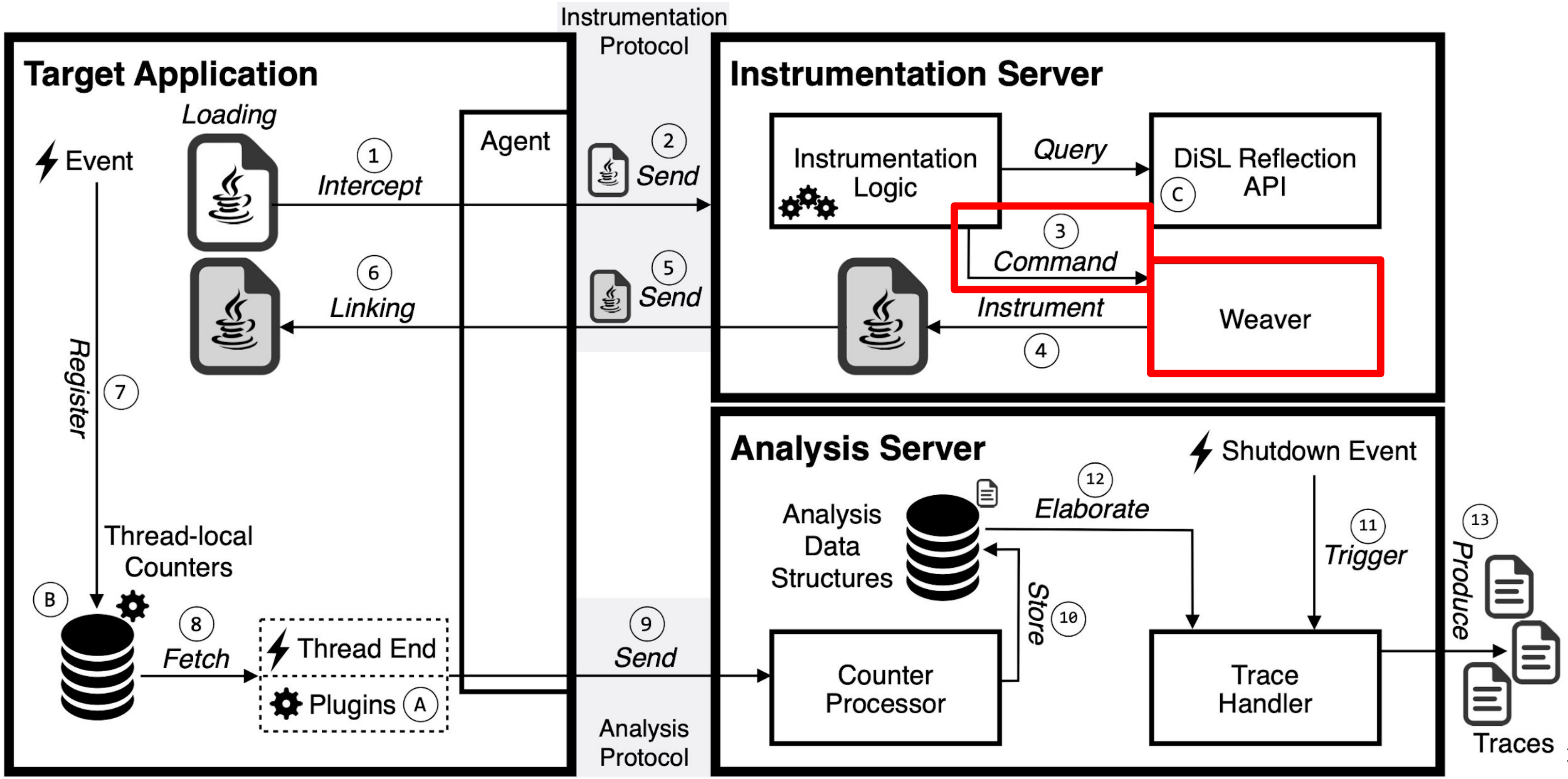
Architecture



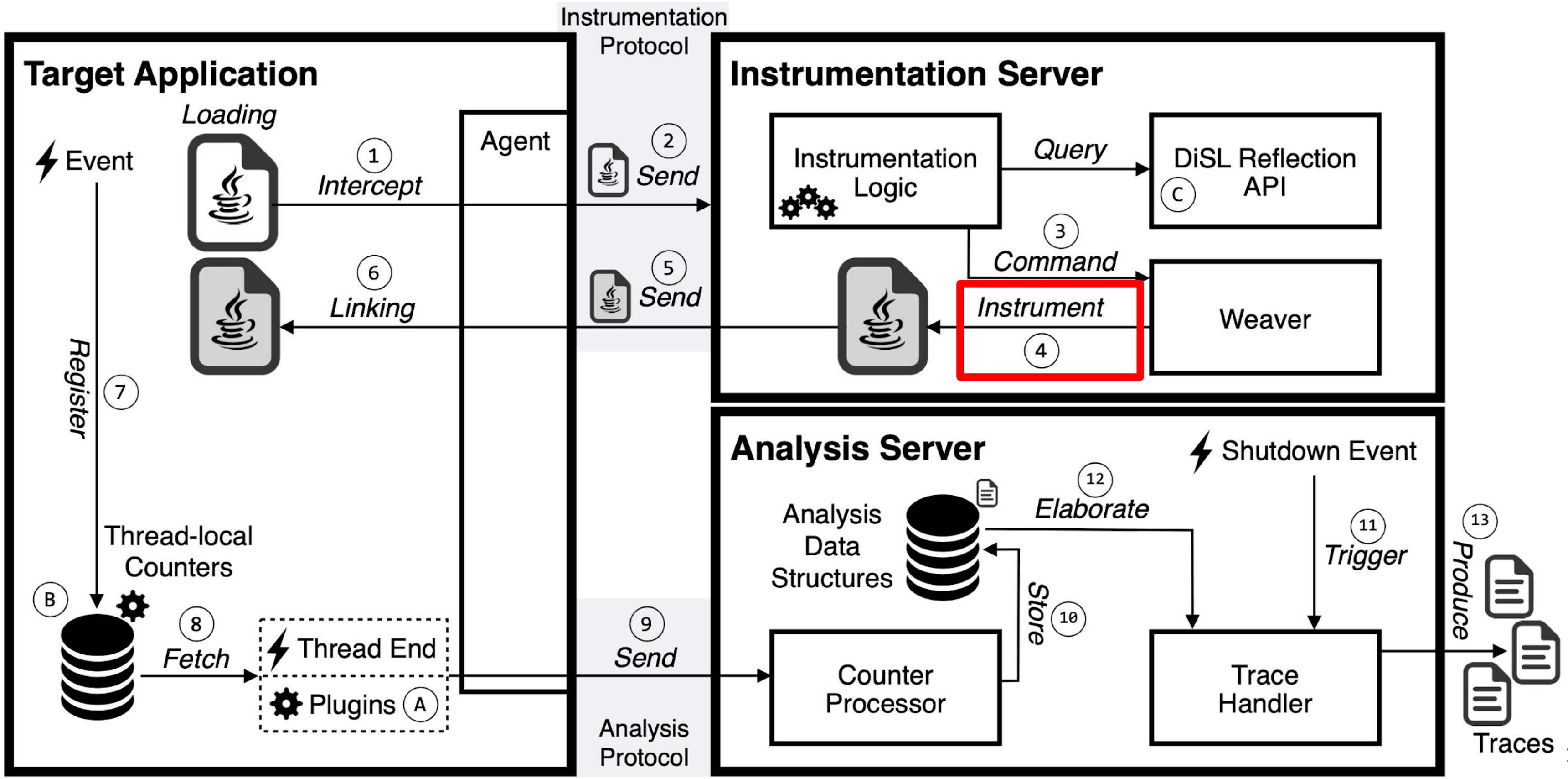
Architecture



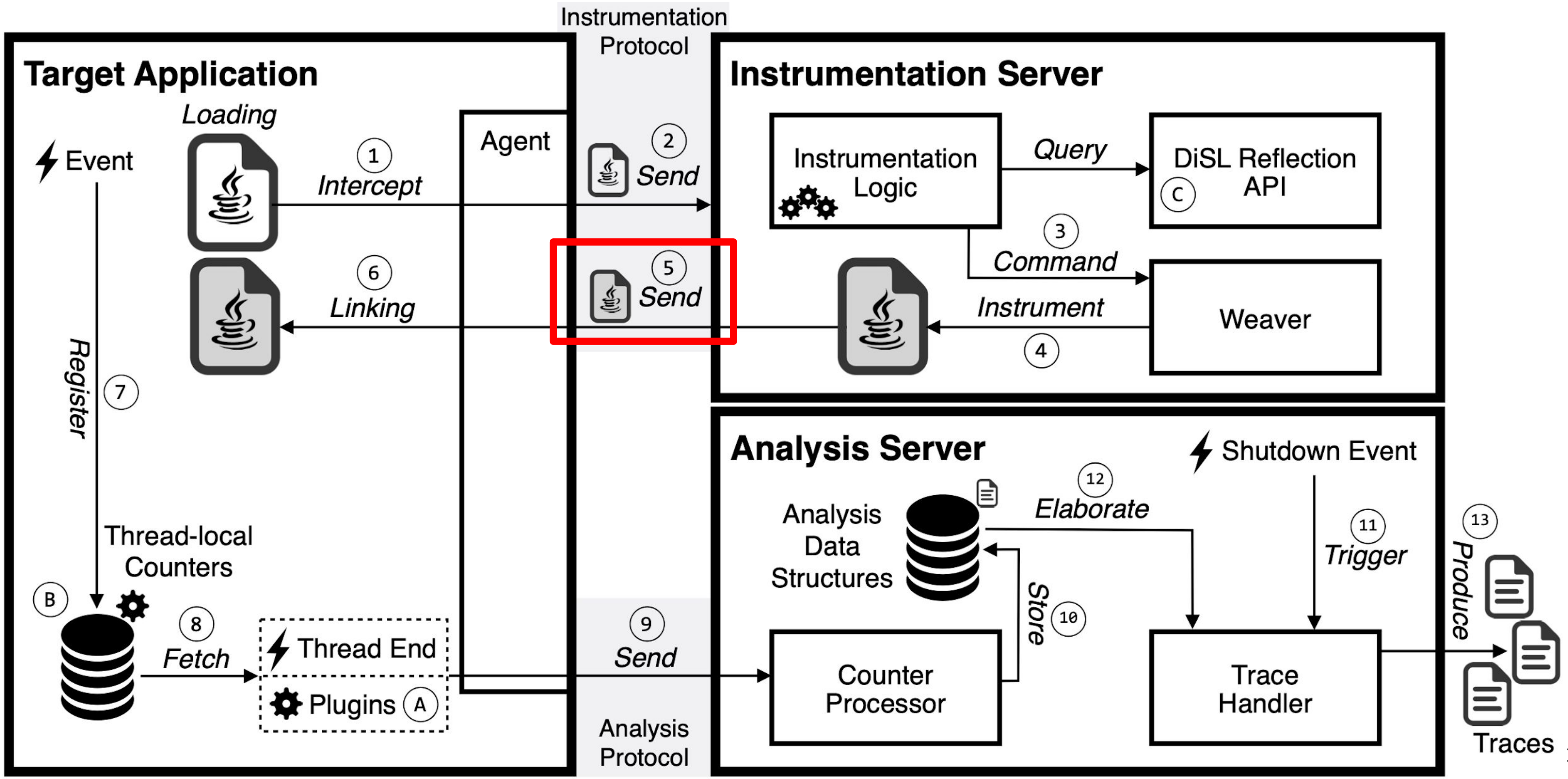
Architecture



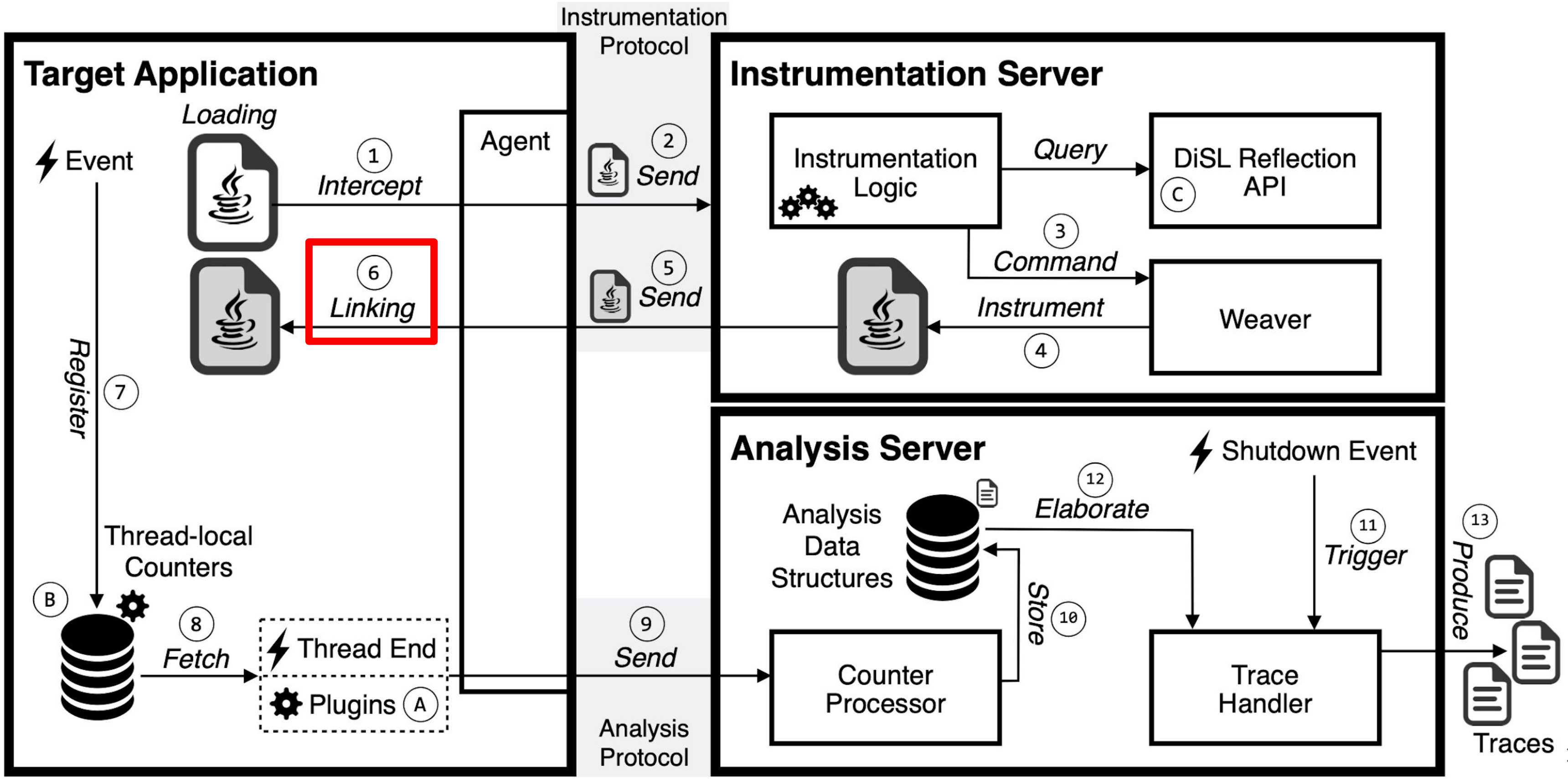
Architecture



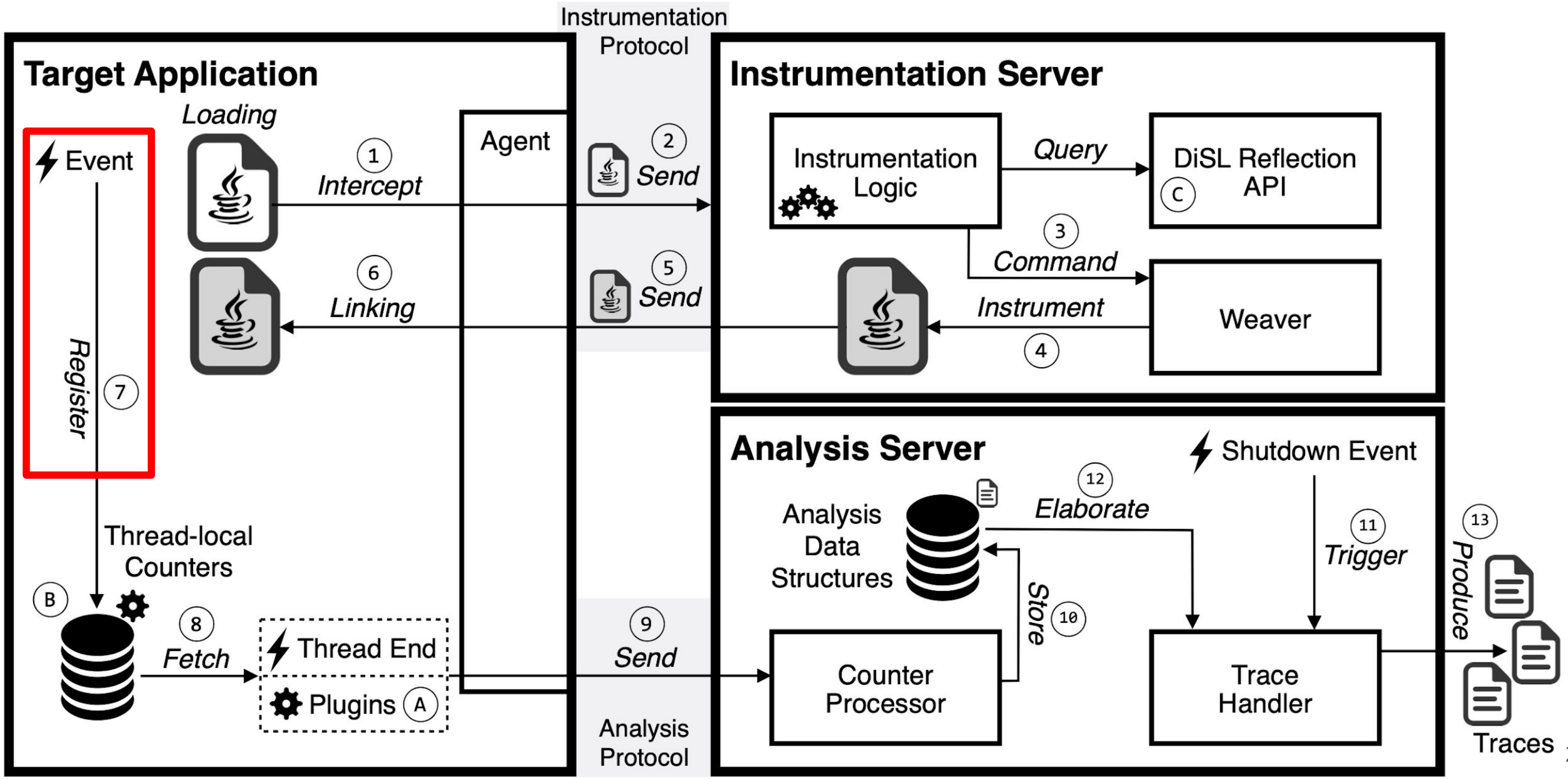
Architecture



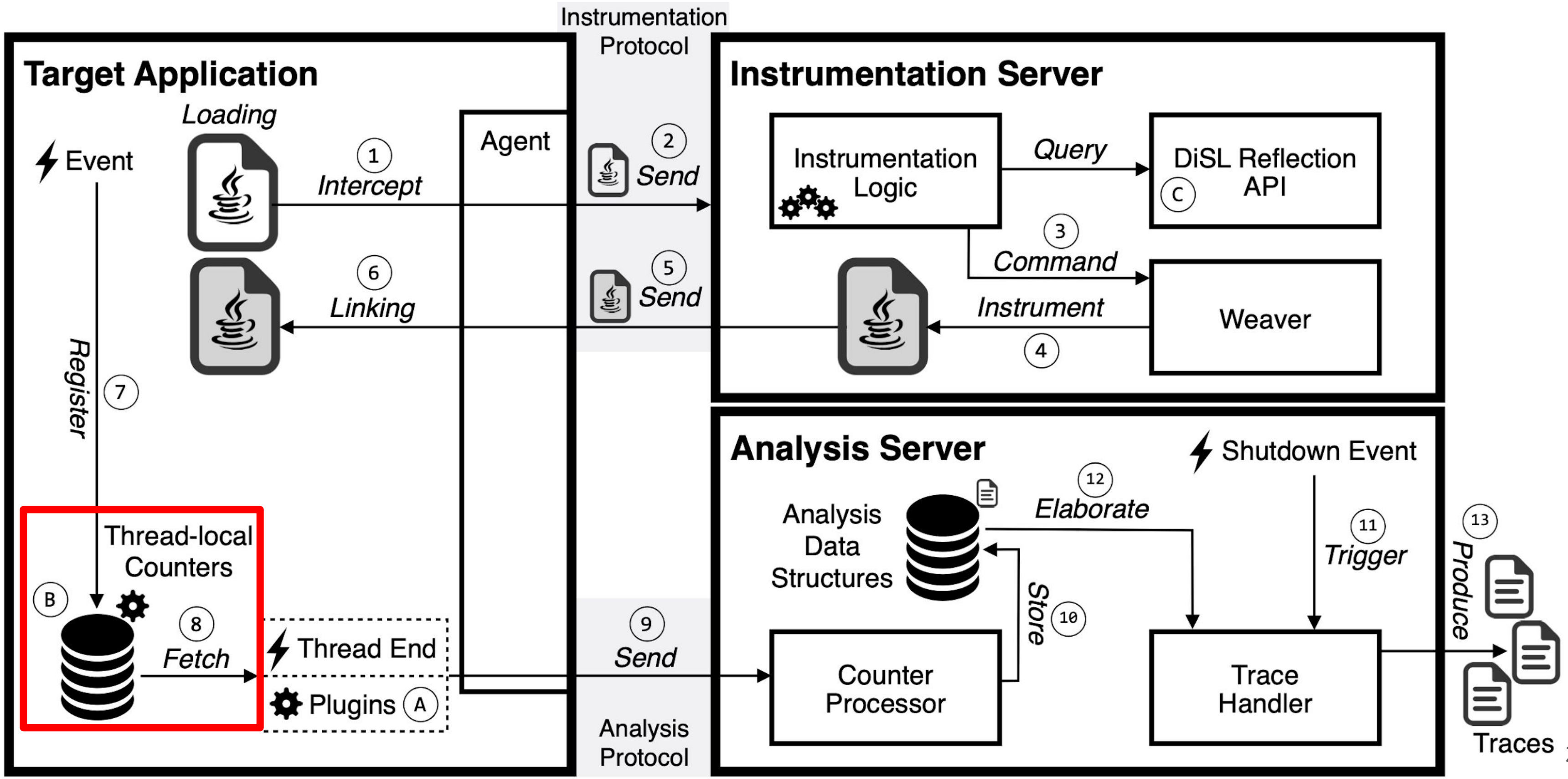
`Architecture



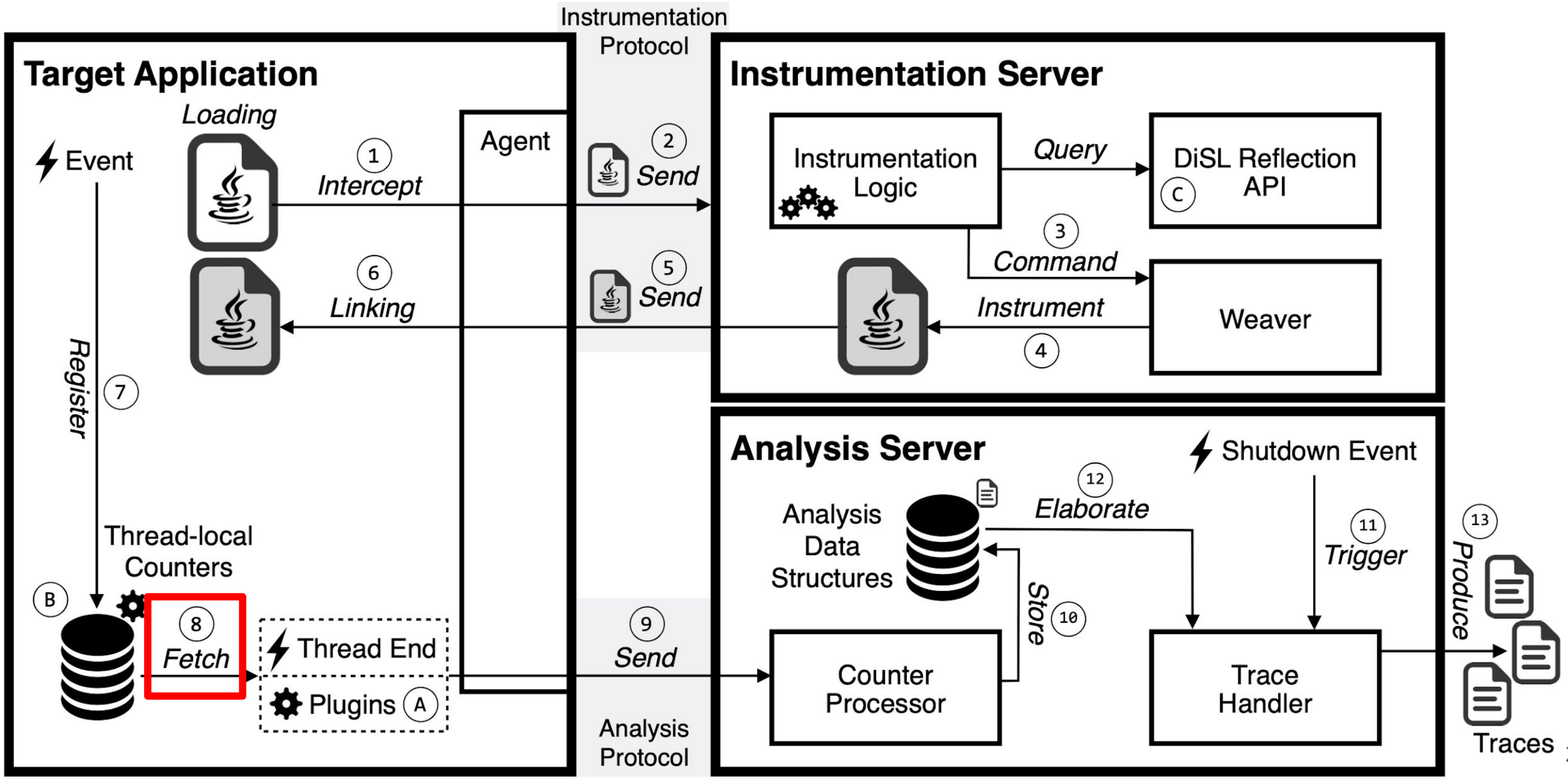
Architecture



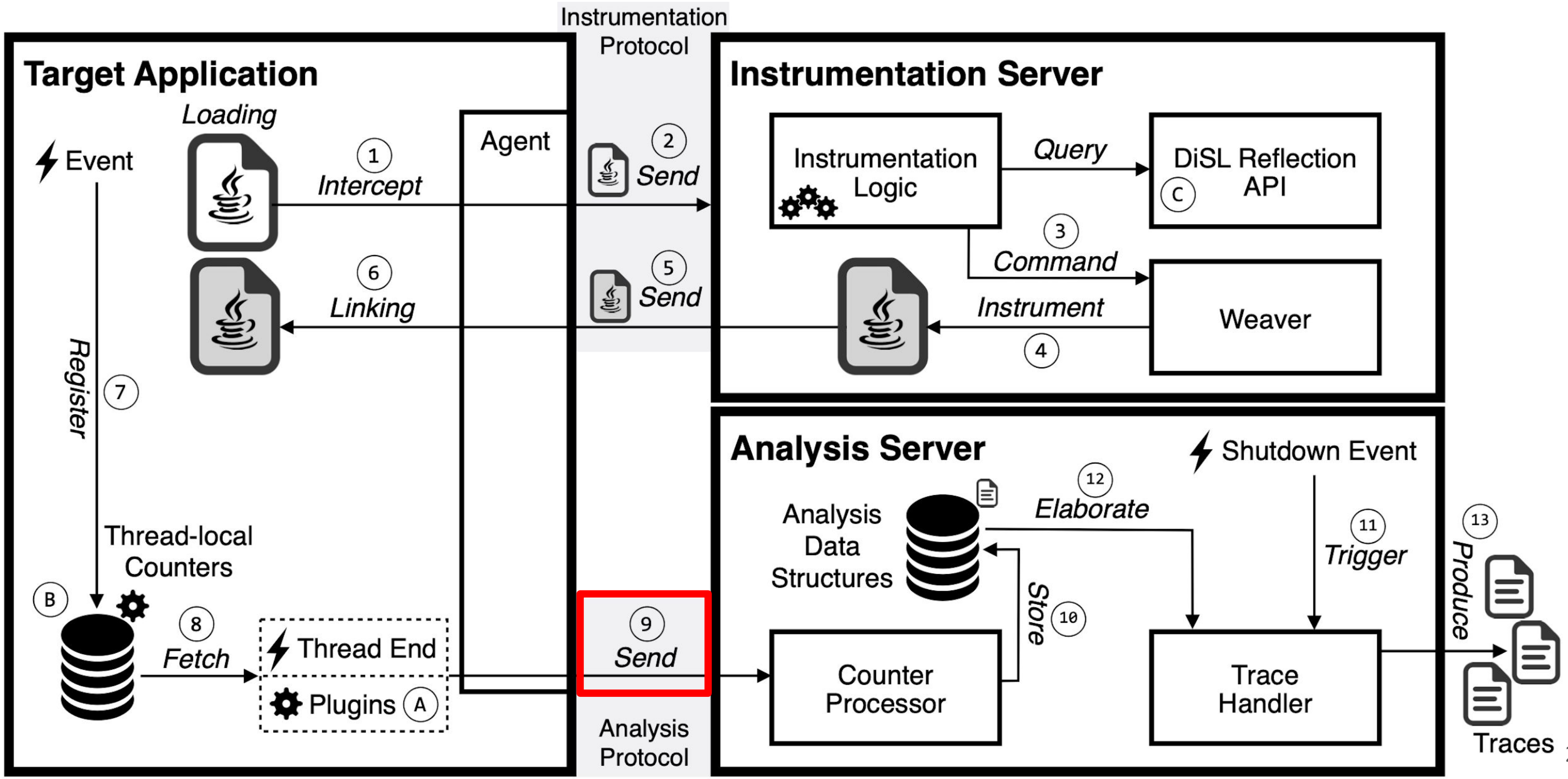
Architecture



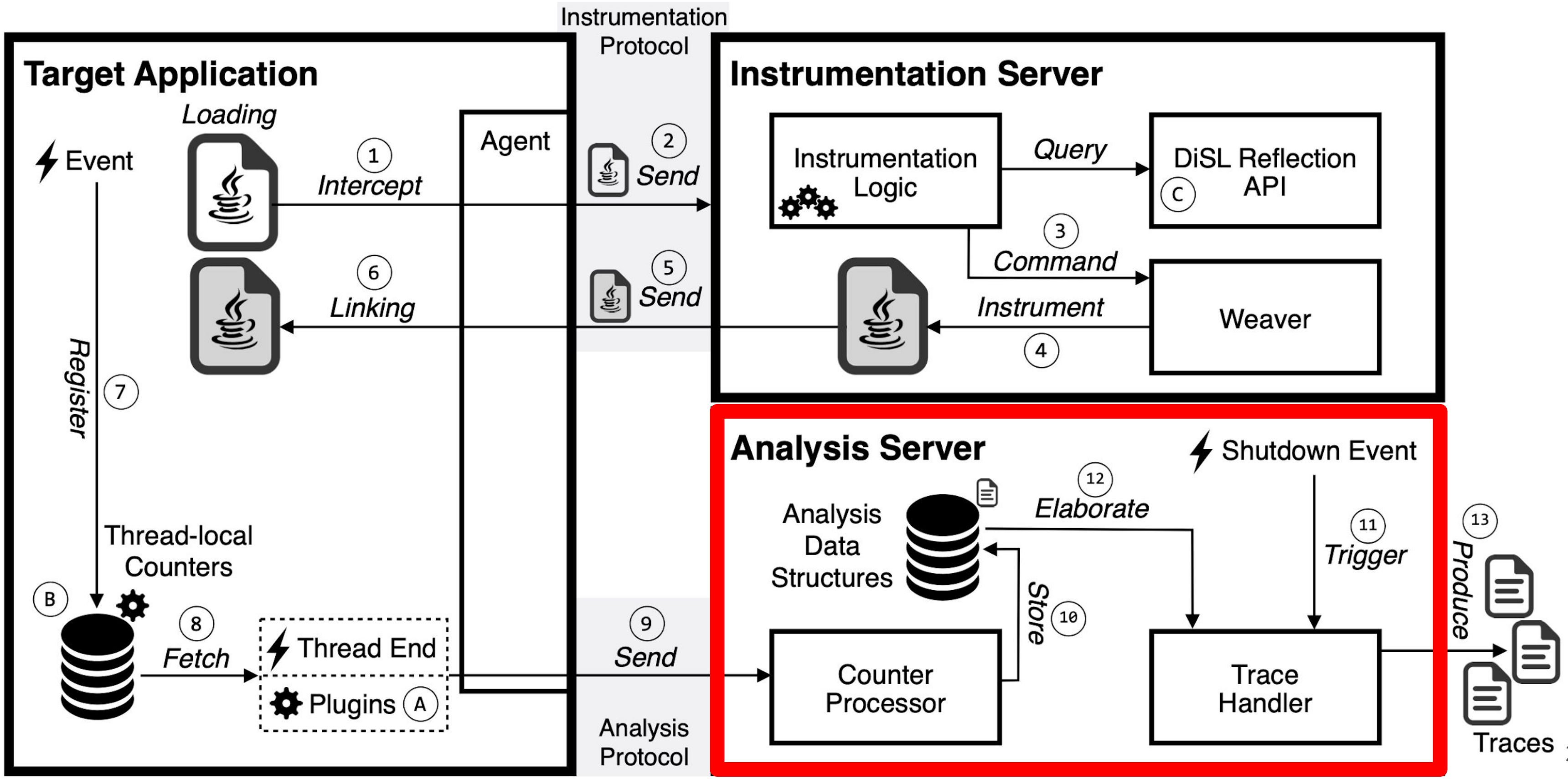
Architecture



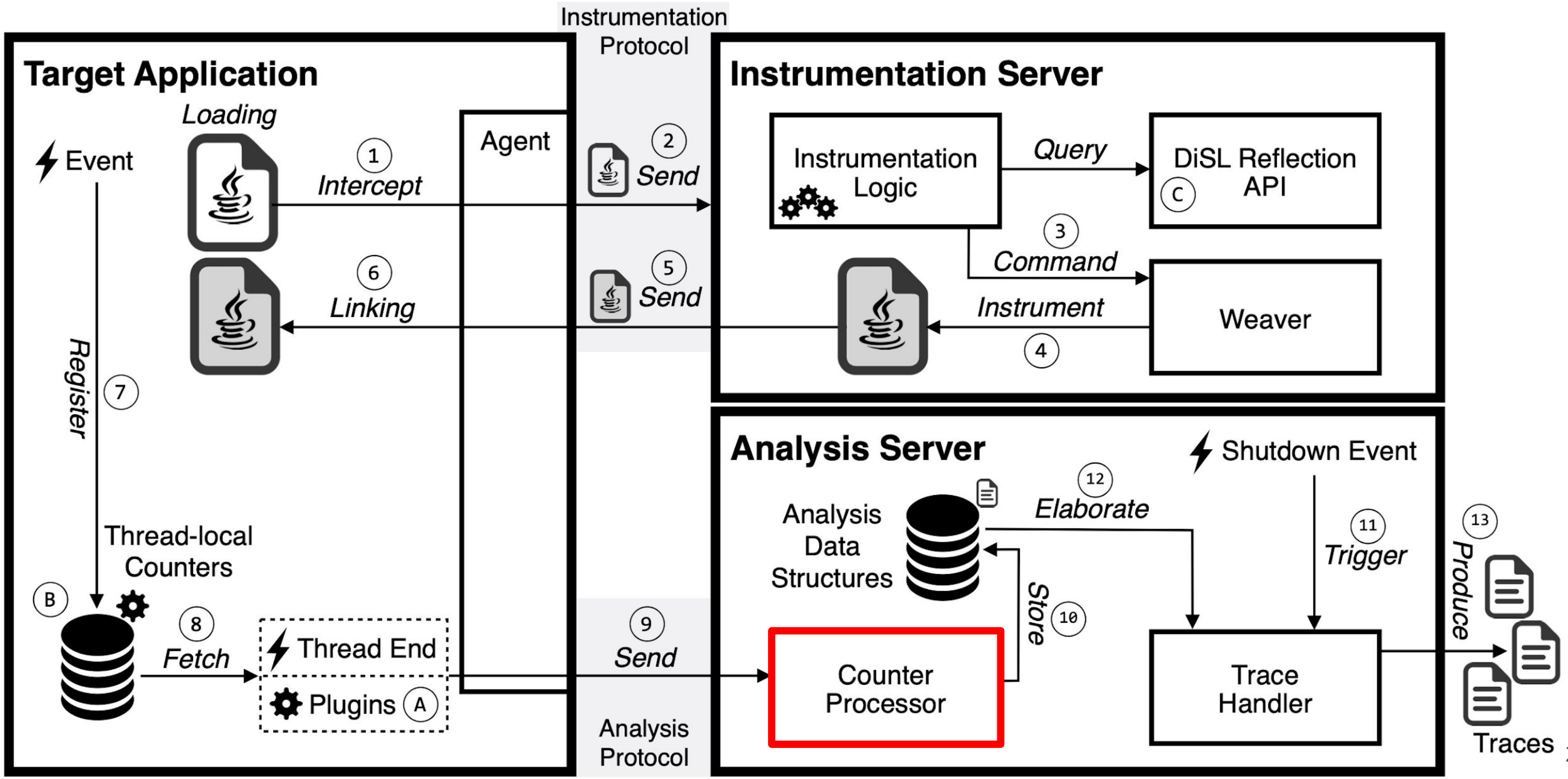
Architecture



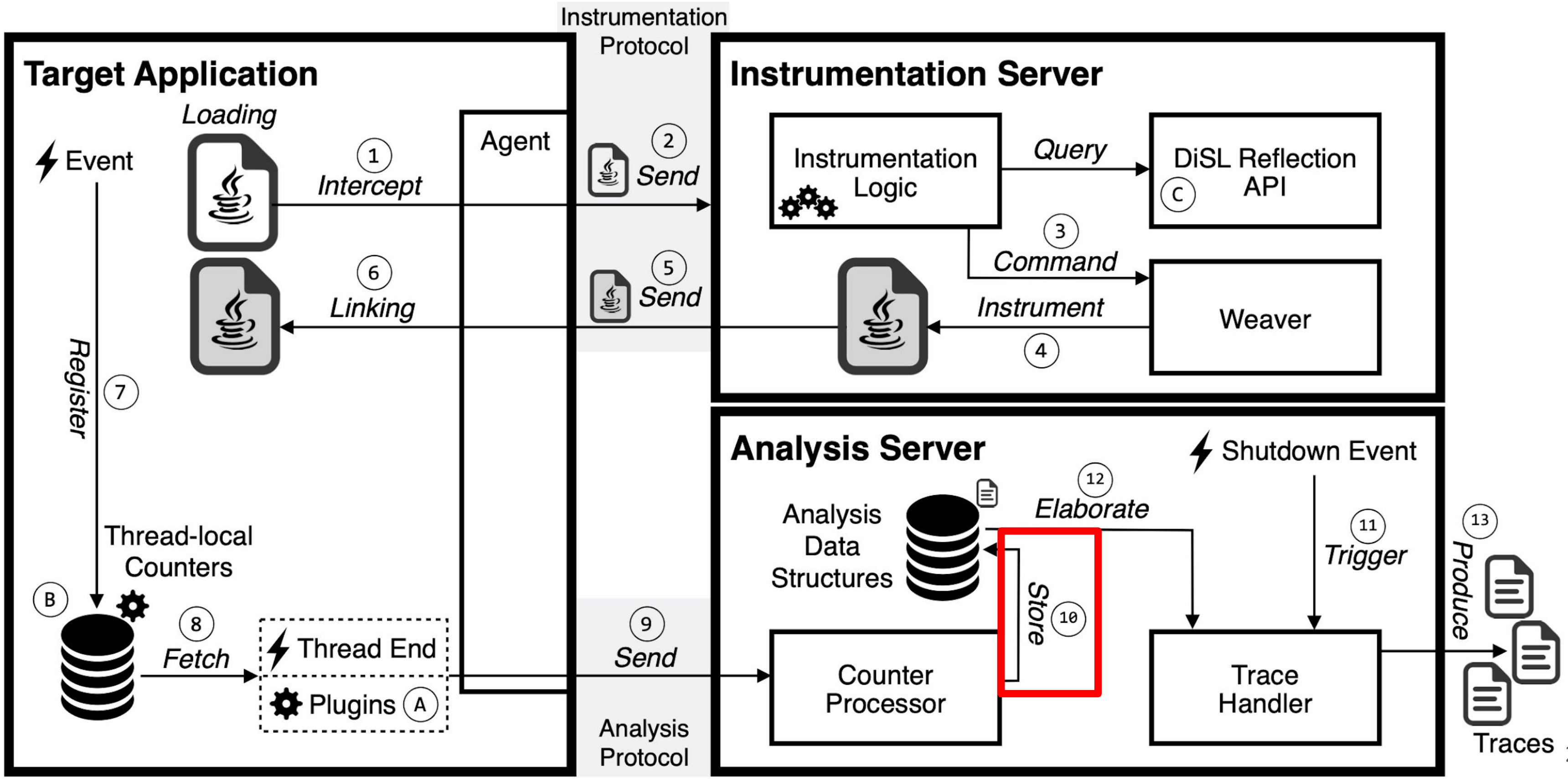
Architecture



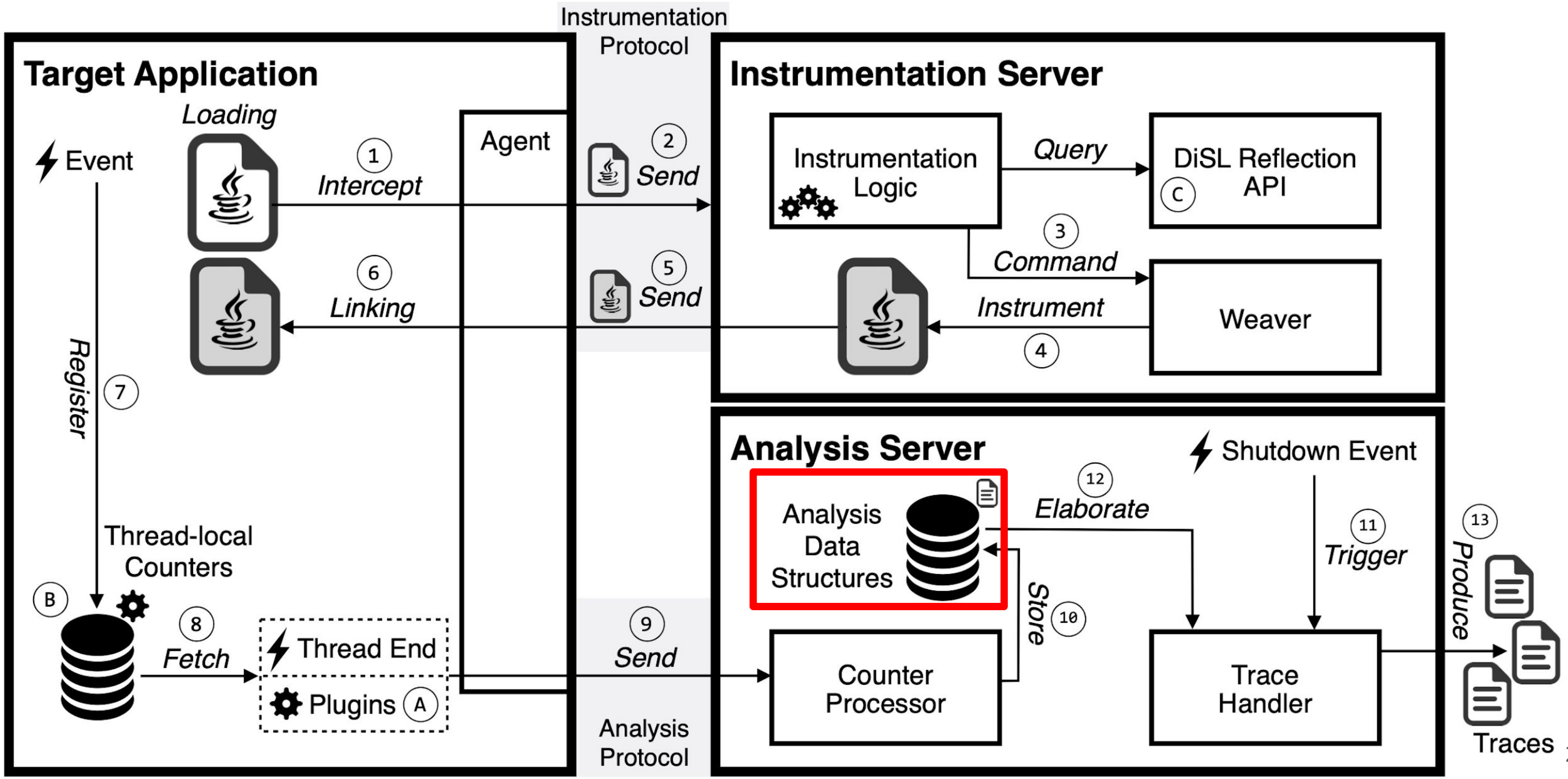
Architecture



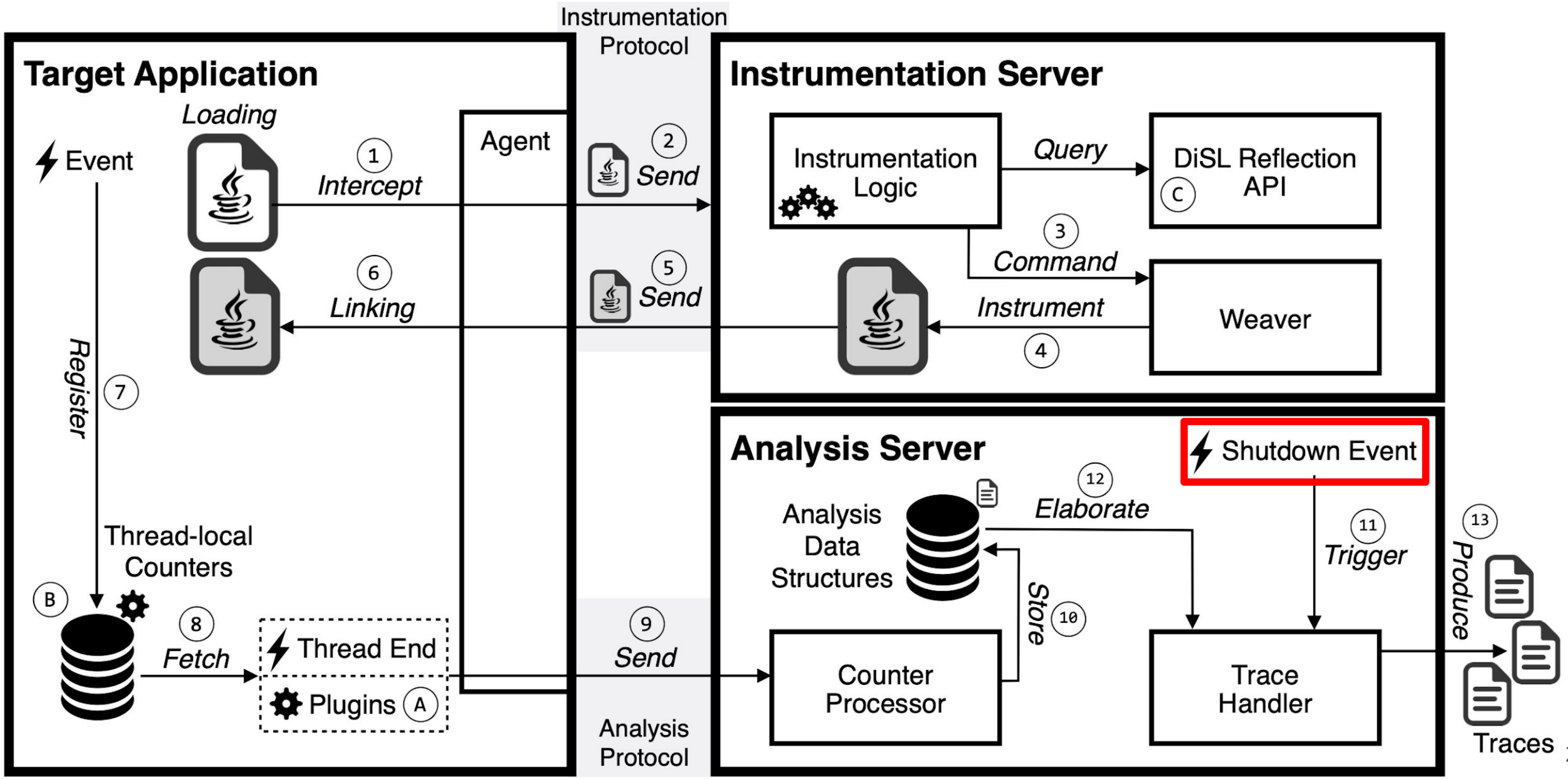
Architecture



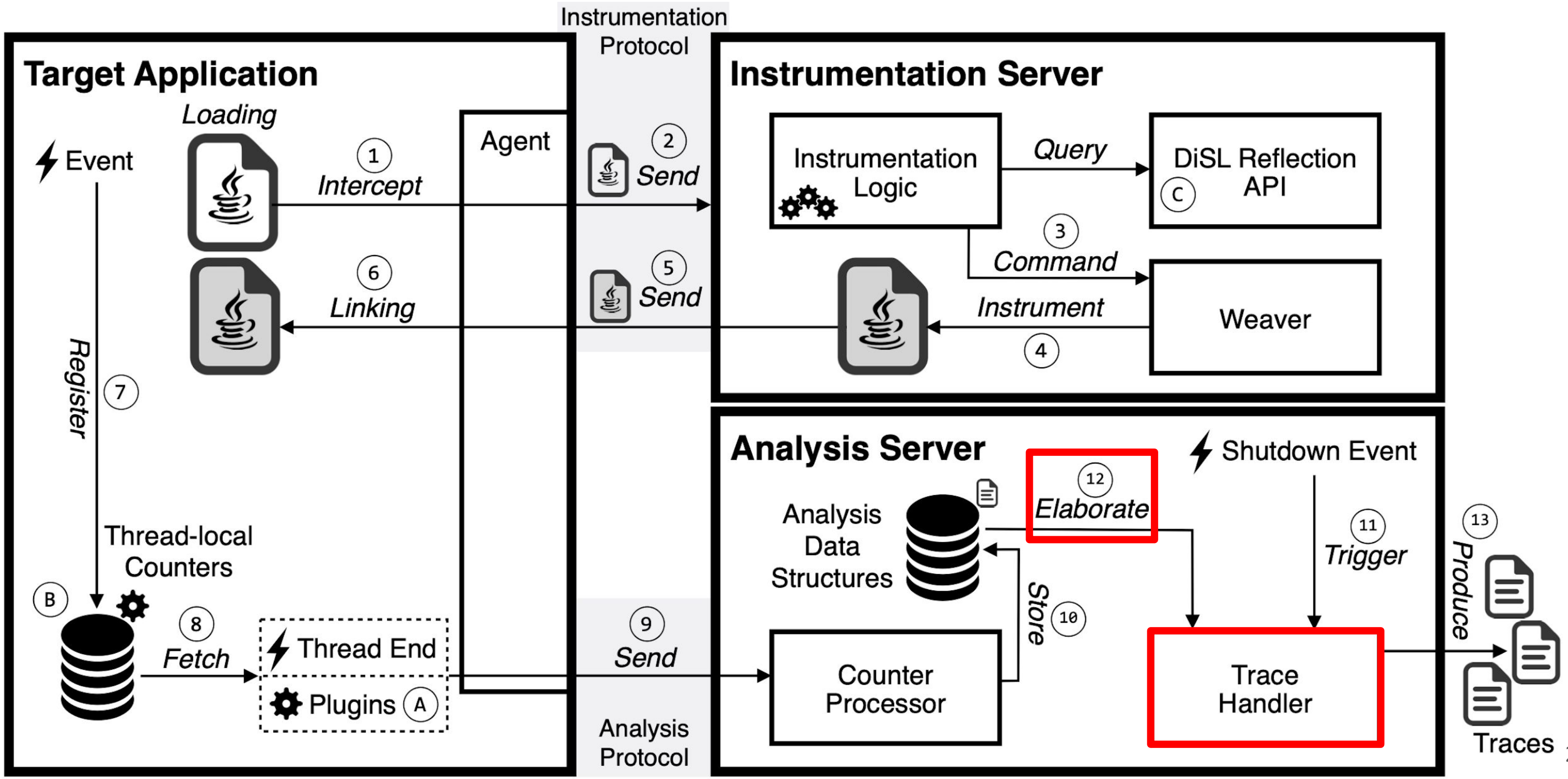
Architecture



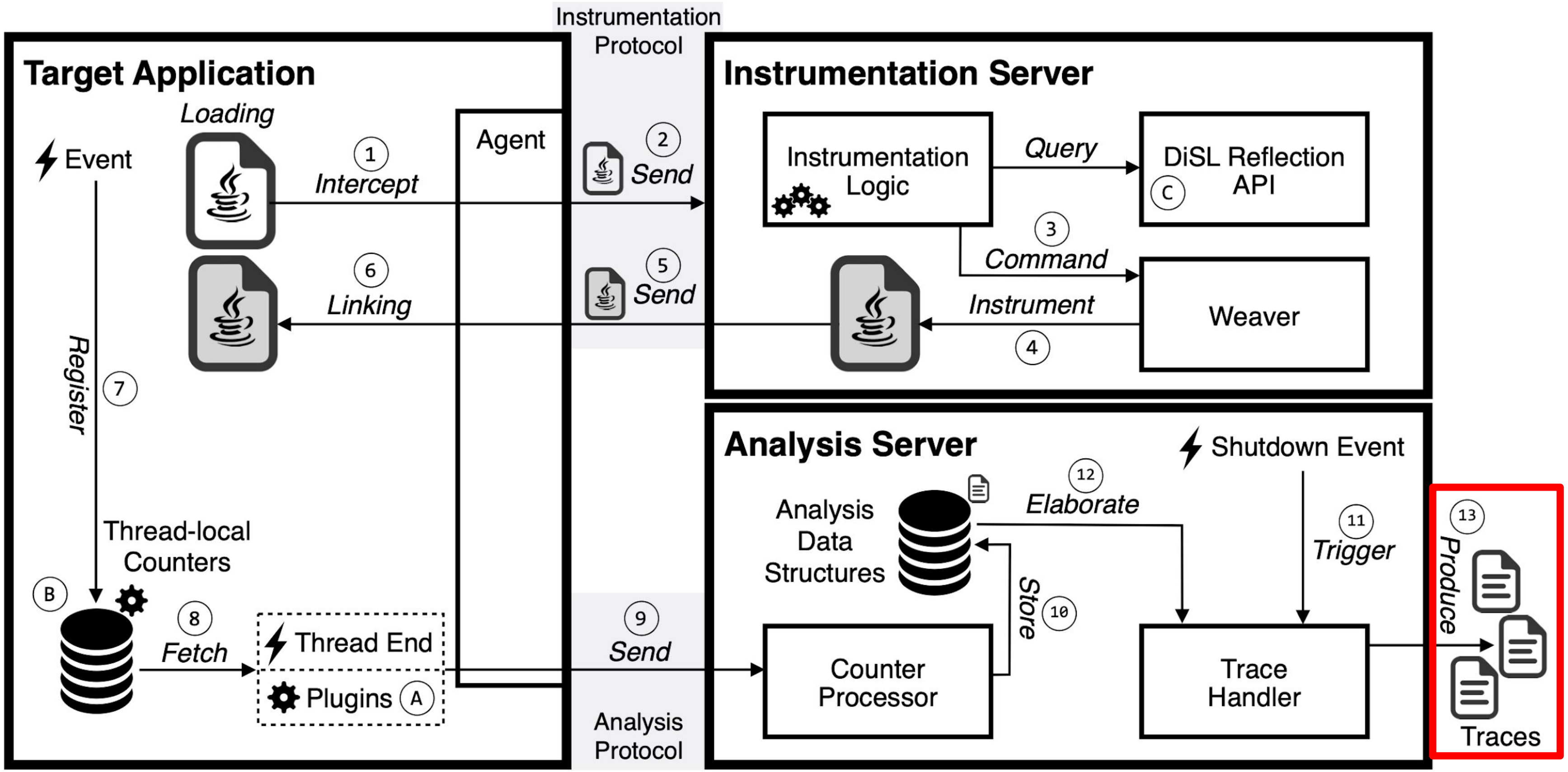
Architecture



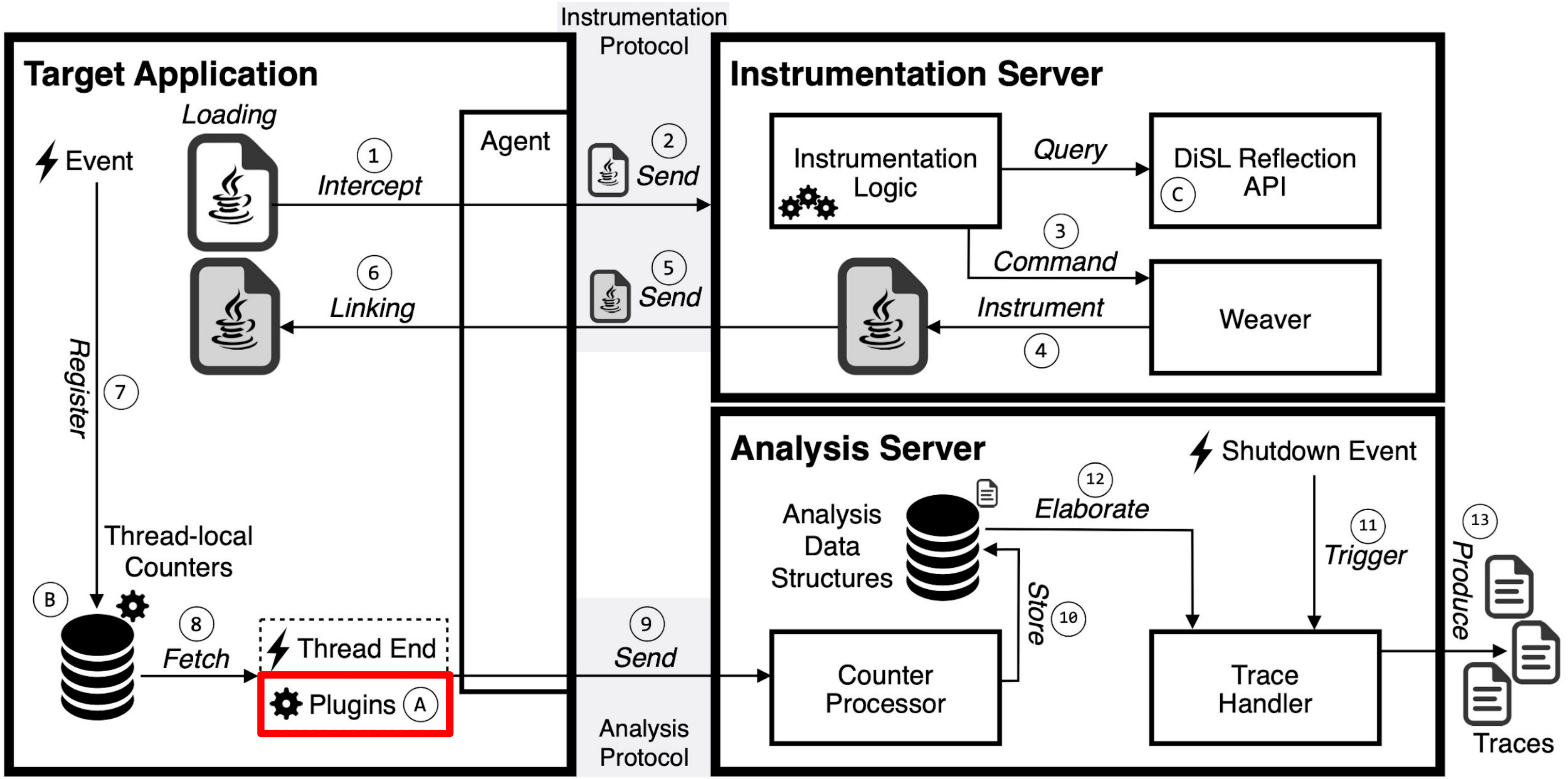
Architecture



Architecture



Architecture



Plugins

- ➢ Allow determining benchmark iteration start/end
  - Useful for differentiating warm-up from steady-state performance
- ➢ P3 includes plugins for Renaissance [1], DaCapo [2], ScalaBench [3], SPECjvm2008 [4]
- ➢ Users can implement plugins for other benchmark suites
- ➢ Can interface with NAB [5]
  - A framework for conducting dynamic analysis on public code repositories

[1] A. Prokopek et al., "Renaissance: Benchmarking Suite for Parallel Applications on the JVM". PLDI 2019.
[2] S. Blackburn et al., "The DaCapo Benchmarks: Java Benchmarking Development and Analysis". SIGPLAN Not. 41(10), 2006.
[3] A. Sewe et al., "Da Capo Con Scala: Design and Analysis of a Scala Benchmark Suite for the Java Virtual Machine". OOPSLA 2011.
[4] SPECjvm2008. `https://www.spec.org/jvm2008/`
[5] A. Villazon et al., "Automated Large-scale Multi-language Dynamic Program Analysis in the Wild". ECOOP 2019.

Main Implementation Details

- ➢ Built on top of the DiSL framework for bytecode instrumentation [1]
  - Guarantees complete bytecode coverage
  - Events of interest detectable also in the Java Class Library
- ➢ Implementation designed to keep profiling overhead moderate while not jeopardizing accuracy
  - Events registered in thread-local counters
    - No synchronization or extra heap allocations
  - Counter elaboration done in a separate process

[1] L. Marek et al., " DiSL: A Domain-specific Language for Bytecode Instrumentation". AOSD 2012.

Applications to Previous Research

- ➢ P^3 was fundamental in development of Renaissance [1]
  - Selection of candidate workloads in public software repositories
    - Showing high concurrency and synchronization
  - Filter out workloads with low parallelism and concurrency
  - Profile key metrics on concurrency and synchronization
    - Demonstrate diversity of Renaissance wrt. other suites
  - Study variability of dynamic metrics across benchmark iterations
- ➢ P^3 was used to conduct large-scale analyses with NAB [2]
  - Particularly on task-parallel workloads

[1] A. Prokopek et al., "Renaissance: Benchmarking Suite for Parallel Applications on the JVM". PLDI 2019.
[2] A. Villazon et al., "Automated Large-scale Multi-language Dynamic Program Analysis in the Wild". ECOOP 2019.

Profiling Overhead

Module	OH
thread	1.00
task	1.03
actor	1.01
future	1.01
ilock	1.03
elock	1.01
wait	1.00
join	1.01
park	1.00
synch	1.01
cas	1.01
atomic	1.01
volatile	1.03
scoll	1.00
ccoll	1.01

➢ Target workload: Renaissance benchmark suite

- Overhead <= 1.01× for most modules
- Overhead = 1.03× for task, ilock and volatile
- Overhead = 1.18× when all modules are active

DEMO

Demo Version

http://dag.inf.usi.ch/software/p3

Limitation

- ➢ Over-profiling possible, if JIT compiler removes events of interest
  - Well-known limitation of bytecode instrumentation

Conclusions

- P^3: a new profiler suite for concurrent applications on the JVM
- Collects many kinds of metrics on parallelism, concurrency and synchronization
- Moderate profiling overhead
- Applicable to prevalent benchmark suites (Renaissance, DaCapo, ScalaBench, SPECjvm2008)
- Suitable for large-scale analysis with NAB
- Fundamental in conducting previous research (e.g., Renaissance)

Thanks for your attention

➢ Contacts:

Andrea Rosà

andrea.rosa@usi.ch