Blkin Code Documentation

End to End tracing in Ceph

Oindrilla Chatterjee[1,a]

Abstract. A brief explanation of the key components of the code which facilitates tracing in Ceph.

Keywords: Blkin, Ceph, Jaeger, tracing.

1. Tracing Infrastructure

Blkin enables tracing in Ceph. It uses the LTTng libraries and tools. Blkin uses Babeltrace plugin to convert captured traces to JSON format and sends them to Zipkin for trace visualization.

Our primary focus would be the Ztracer Namespace in Ceph.

Directory: blkin/blkin-lib/ztracer.hpp

```
#ifndef ZTRACER_H

#define ZTRACER_H

#include <string>
extern "C" {
#include <zipkin_c.h>
}

namespace ZTracer {
    using std::string;

    const char* const CLIENT_SEND = "cs";
    const char* const CLIENT_RECV = "cr";
    const char* const SERVER_SEND = "ss";
    const char* const SERVER_RECV = "sr";
    const char* const WIRE_SEND = "ws";
    const char* const WIRE_RECV = "wr";
    const char* const CLIENT_SEND_FRAGMENT = "csf";
    const char* const CLIENT_RECV_FRAGMENT = "crf";
    const char* const SERVER_SEND_FRAGMENT = "ssf";
    const char* const SERVER_RECV_FRAGMENT = "srf";
```

[1]College of Engineering, Boston University, Boston, US
[a]oc@bu.edu

```
static inline int ztrace_init() { return blkin_init(); }

class Endpoint : private blkin_endpoint {
private:
    string _ip; // storage for blkin_endpoint.ip, see copy_ip()
    string _name; // storage for blkin_endpoint.name, see copy_name()

    friend class Trace;
public:
    Endpoint(const char *name)
        {
            blkin_init_endpoint(this, "0.0.0.0", 0, name);
        }
    Endpoint(const char *ip, int port, const char *name)
        {
            blkin_init_endpoint(this, ip, port, name);
        }

    // copy constructor and operator need to account for ip/name storage
    Endpoint(const Endpoint &rhs) : _ip(rhs._ip), _name(rhs._name)
        {
            blkin_init_endpoint(this, _ip.size() ? _ip.c_str() : rhs.ip,
                                rhs.port,
                                _name.size() ? _name.c_str(): rhs.name);
        }
    const Endpoint& operator=(const Endpoint &rhs)
        {
            _ip.assign(rhs._ip);
            _name.assign(rhs._name);
            blkin_init_endpoint(this, _ip.size() ? _ip.c_str() : rhs.ip,
                                rhs.port,
                                _name.size() ? _name.c_str() : rhs.name);
            return *this;
        }

    // Endpoint assumes that ip and name will be string literals, and
    // avoids making a copy and freeing on destruction.  if you need to
    // initialize Endpoint with a temporary string, copy_ip/copy_name()
    // will store it in a std::string and assign from that
    void copy_ip(const string &newip)
        {
            _ip.assign(newip);
            ip = _ip.c_str();
        }
```

```
    void copy_name(const string &newname)
        {
            _name.assign(newname);
            name = _name.c_str();
        }

    void copy_address_from(const Endpoint *endpoint)
        {
            _ip.assign(endpoint->ip);
            ip = _ip.c_str();
            port = endpoint->port;
        }
    void share_address_from(const Endpoint *endpoint)
        {
            ip = endpoint->ip;
            port = endpoint->port;
        }
    void set_port(int p) { port = p; }
};

class Trace : private blkin_trace {
private:
    string _name; // storage for blkin_trace.name, see copy_name()

public:
    // default constructor zero-initializes blkin_trace valid()
    // will return false on a default-constructed Trace until
    // init()
    Trace()
        {
            // zero-initialize so valid() returns false
            name = NULL;
            info.trace_id = 0;
            info.span_id = 0;
            info.parent_span_id = 0;
            endpoint = NULL;
        }

    // construct a Trace with an optional parent
    Trace(const char *name, const Endpoint *ep, const Trace *parent = NULL)
        {
            if (parent && parent->valid()) {
                blkin_init_child(this, parent, ep ? : parent->endpoint,
                                 name);
            } else {
```

```
            blkin_init_new_trace(this, name, ep);
        }
    }

// construct a Trace from blkin_trace_info
Trace(const char *name, const Endpoint *ep,
      const blkin_trace_info *i, bool child=false)
    {
        if (child)
            blkin_init_child_info(this, i, ep, name);
        else {
            blkin_init_new_trace(this, name, ep);
            set_info(i);
        }
    }

// copy constructor and operator need to account for name storage
Trace(const Trace &rhs) : _name(rhs._name)
    {
        name = _name.size() ? _name.c_str() : rhs.name;
        info = rhs.info;
        endpoint = rhs.endpoint;
    }
const Trace& operator=(const Trace &rhs)
    {
        _name.assign(rhs._name);
        name = _name.size() ? _name.c_str() : rhs.name;
        info = rhs.info;
        endpoint = rhs.endpoint;
        return *this;
    }

// return true if the Trace has been initialized
bool valid() const { return info.trace_id != 0; }
operator bool() const { return valid(); }

// (re)initialize a Trace with an optional parent
int init(const char *name, const Endpoint *ep,
         const Trace *parent = NULL)
    {
        if (parent && parent->valid())
            return blkin_init_child(this, parent,
                                    ep ? : parent->endpoint, name);

        return blkin_init_new_trace(this, name, ep);
```

```
    }

// (re)initialize a Trace from blkin_trace_info
int init(const char *name, const Endpoint *ep,
         const blkin_trace_info *i, bool child=false)
    {
        if (child)
            return blkin_init_child_info(this, i, ep, _name.c_str());

        return blkin_set_trace_properties(this, i, _name.c_str(), ep);
    }

// Trace assumes that name will be a string literal, and avoids
// making a copy and freeing on destruction.  if you need to
// initialize Trace with a temporary string, copy_name() will store
// it in a std::string and assign from that
void copy_name(const string &newname)
    {
        _name.assign(newname);
        name = _name.c_str();
    }

const blkin_trace_info* get_info() const { return &info; }
void set_info(const blkin_trace_info *i) { info = *i; }

// record key-value annotations
void keyval(const char *key, const char *val) const
    {
        if (valid())
            BLKIN_KEYVAL_STRING(this, endpoint, key, val);
    }
void keyval(const char *key, int64_t val) const
    {
        if (valid())
            BLKIN_KEYVAL_INTEGER(this, endpoint, key, val);
    }
void keyval(const char *key, const char *val, const Endpoint *ep) const
    {
        if (valid())
            BLKIN_KEYVAL_STRING(this, ep, key, val);
    }
void keyval(const char *key, int64_t val, const Endpoint *ep) const
    {
        if (valid())
            BLKIN_KEYVAL_INTEGER(this, ep, key, val);
```

```
            }

        // record timestamp annotations
        void event(const char *event) const
            {
                if (valid())
                    BLKIN_TIMESTAMP(this, endpoint, event);
            }
        void event(const char *event, const Endpoint *ep) const
            {
                if (valid())
                    BLKIN_TIMESTAMP(this, ep, event);
            }
    };

}
#endif /* end of include guard: ZTRACER_H */
```

The class Endpoint is responsible for identifying the component that is being traced. It defines member variables which describe the Port number, IP Address and Service name of a component.

The class Trace is responsible for defining variables pertaining to a trace which refer to the Trace ID, Span ID, Parent Span ID etc. The functions in the Trace class, most importantly init(), event() are responsible for carrying out a trace.

2. Tracepoints in Ceph code

Tracing is carried out in Ceph in a following manner.

Directory: src/librbd/LibrbdWriteback.cc

```
ZTracer::Trace trace;
if (parent_trace.valid()) {
  trace.init("", &m_ictx->trace_endpoint, &parent_trace);
  trace.copy_name("cache read " + oid.name);
  trace.event("start");
}
```

Directory: src/librbd/io/ImageRequestWQ.cc

```
ZTracer::Trace trace;
if (m_image_ctx.blkin_trace_all) {
 trace.init("wq: flush", &m_image_ctx.trace_endpoint);
 trace.event("init");
}
```

trace.init() and trace.event() are the root functions which facilitate tracing in Ceph.

3. init() function

The init() function initializes a trace, specifically initializes the IDs either with default parameters if it is a root trace, else with parent span ID of the parent.

Directory: blkin-lib/ztracer.hpp

```
int init(const char *name, const Endpoint *ep,
                 const Trace *parent = NULL)
            {
                if (parent && parent->valid())
                    return blkin_init_child(this, parent,
                                            ep ? : parent->endpoint, name);

                return blkin_init_new_trace(this, name, ep);
            }

// (re)initialize a Trace from blkin_trace_info

int init(const char *name, const Endpoint *ep,
                 const blkin_trace_info *i, bool child=false)
            {
                if (child)
                    return blkin_init_child_info(this, i, ep, _name.c_str());

                return blkin_set_trace_properties(this, i, _name.c_str(), ep);
            }
```

4. event() function

The event() function is responsible for tracing an event. We will look into the function calls this function leads to. The body of the event function is as follows.
Directory: blkin-lib/ztracer.hpp

```
void event(const char *event) const
    {
   if (valid())
       BLKIN_TIMESTAMP(this, endpoint, event);
    }
void event(const char *event, const Endpoint *ep) const
    {
```

```
   if (valid())
       BLKIN_TIMESTAMP(this, ep, event);
    }
   };
```

4.1. Arguments

The argument of event function is a character string [event() is invoked as event("start"), event("finish"), event("created trace"), event("decoded trace")].

event() function calls BLKIN_TIMESTAMP macro with the following arguments.
1. **trace**(Ztracer::Trace trace which is an object of class Trace in Ztracer namespace. trace object is defined before each invocation of trace.event("start") [eg. check in Directory : ceph/src/osdc/ObjectCacher.cc].
2. **endpoint**(a member variable of Trace class defined in blkin-lib/ztracer.hpp).
3. **event** (character string which is passed on to it during invokation).

Directory: blkin-lib/ztracer.hpp

```
class Trace : private blkin_trace {
   private:
string _name; // storage for blkin_trace.name, see copy_name()

   public:
// default constructor zero-initializes blkin_trace valid()
// will return false on a default-constructed Trace until
// init()
Trace()
    {
   // zero-initialize so valid() returns false
   name = NULL;
   info.trace_id = 0;
   info.span_id = 0;
   info.parent_span_id = 0;
   endpoint = NULL;
    }
```

4.2. Body

The body of the event() function consists of calling the BLKIN_TIMESTAMP macro which is defined as follows.

Directory:blkin-lib/zipkin_c.h

```
#define BLKIN_TIMESTAMP(trace, endp, event)
   do {
      struct blkin_annotation __annot;
      blkin_init_timestamp_annotation(&__annot, event, endp);
      blkin_record(trace, &__annot);
   } while (0);
```

4.3. BLKIN_TIMESTAMP

The arguments of BLKIN_TIMESTAMP have been described above. The body of BLKIN_TIMESTAMP is as follows.
It calls two functions
1. blkin_init_timestamp_annotation
2. blkin_record

These functions would be described in the following few lines.

4.4. blkin_init_timestamp_annotation

i. blkin_init_timestamp_annotation is defined as follows.

Directory: blkin-lib/zipkin_c.c

```
int blkin_init_timestamp_annotation(struct blkin_annotation *annotation,
                const char *event, const struct blkin_endpoint *endpoint)
{
   int res;
   if ((!annotation) || (!event)){
      res = -EINVAL;
      goto OUT;
   }
   annotation->type = ANNOT_TIMESTAMP;
   annotation->val_str = event;
   annotation->endpoint = endpoint;
   res = 0;

OUT:
   return res;
}
```

The arguments of this function are described as follows.
1. **annotation**(pointer to the structure blkin_annotation)
2. '**event**' string which we passed on to event() as an argument
3. **endpoint**(pointer to struct blkin_endpoint)

Directory: blkin-lib/zipkin_c.c

```
struct blkin_annotation {
    blkin_annotation_type type;
    const char *key;
    union {
   const char *val_str;
   int64_t val_int;
    };
    const struct blkin_endpoint *endpoint;
};
```

Directory: blkin-lib/zipkin_c.h

```
struct blkin_endpoint {
    const char *ip;
    int16_t port;
    const char *name;
blkin-lib/zipkin_c.h
```

The body of the function can be described as follows.
It assigns annotation type, its val_str and endpoint to variables of the blkin_annotation structure.
1. type = ANNOT_TIMESTAMP
2. val_str = event [string which we had passed on as an argument to event()]
3. blkin_endpoint = endpoint

Directory: blkin-lib/zipkin_c.h

```
typedef enum {
    ANNOT_STRING = 0,
    ANNOT_INTEGER,
    ANNOT_TIMESTAMP
} blkin_annotation_type;
```

4.5. blkin_record()

ii. blkin_record is defined as follows.

Directory: blkin-lib/zipkin_c.c

```
int blkin_record(const struct blkin_trace *trace,
       const struct blkin_annotation *annotation)
{
   int res;
   if (!annotation || !trace || !trace->name) {
      res = -EINVAL;
```

```
      goto OUT;
   }

   const struct blkin_endpoint *endpoint =
      annotation->endpoint ? : trace->endpoint;
   if (!endpoint || !endpoint->ip || !endpoint->name) {
      res = -EINVAL;
      goto OUT;
   }

   if (annotation->type == ANNOT_STRING) {
      if ((!annotation->key) || (!annotation->val_str)) {
         res = -EINVAL;
         goto OUT;
      }
/*moc: look for equivalent function in Jaeger and see how it can be equated */

      tracepoint(zipkin, keyval_string, trace->name,
            endpoint->name, endpoint->port, endpoint->ip,
            trace->info.trace_id, trace->info.span_id,
            trace->info.parent_span_id,
            annotation->key, annotation->val_str);
   }
   else if (annotation->type == ANNOT_INTEGER) {
      if (!annotation->key) {
         res = -EINVAL;
         goto OUT;
      }
/*moc: look for equivalent function in Jaeger and see how it can be equated */
      tracepoint(zipkin, keyval_integer, trace->name,
            endpoint->name, endpoint->port, endpoint->ip,
            trace->info.trace_id, trace->info.span_id,
            trace->info.parent_span_id,
            annotation->key, annotation->val_int);
   }
   else {
      if (!annotation->val_str) {
         res = -EINVAL;
         goto OUT;
      }

/*moc: look for equivalent function in Jaeger and see how it can be equated */
      tracepoint(zipkin, timestamp , trace->name,
            endpoint->name, endpoint->port, endpoint->ip,
            trace->info.trace_id, trace->info.span_id,
```

```
            trace->info.parent_span_id,
            annotation->val_str);
    }
    res = 0;
OUT:
    return res;
}
```

The arguments are described as follows.
1. **trace** is an object of blkin_trace.
2. **annotation**(pointer to the structure blkin_annotation)

Directory: blkin-lib/zipkin_c.h

```
struct blkin_trace {
    const char *name;
    struct blkin_trace_info info;
    const struct blkin_endpoint *endpoint;
};
```

4.6. Body of record()

The body of the record() function is described as follows.

I. If annotation-type == ANNOT_STRING
We call the tracepoint() function with the following arguments.
1. **zipkin**
Directory: babeltrace-plugins/zipkin/zipkin_trace.h

```
#define TRACEPOINT_PROVIDER zipkin
```

2. **keyval_string**
Directory: blkin-lib/zipkin_trace.h

```
TRACEPOINT_EVENT(
    zipkin,
    keyval_string,
    TP_ARGS(const char *, trace_name, const char *, service,
        int, port, const char *, ip, long, trace,
        long, span, long, parent_span,
        const char *, key, const char *, val ),

    TP_FIELDS(
        /*
         * Each span has a name mentioned on it in the UI
         * This is the trace name
         */
```

```
        ctf_string(trace_name, trace_name)
        /*
         * Each trace takes place in a specific machine-endpoint
         * This is identified by a name, a port number and an ip
         */
      ctf_string(service_name, service)
      ctf_integer(int, port_no, port)
      ctf_string(ip, ip)
      /*
       * According to the tracing semantics each trace should have
       * a trace id, a span id and a parent span id
       */
      ctf_integer(long, trace_id, trace)
      ctf_integer(long, span_id, span)
      ctf_integer(long, parent_span_id, parent_span)
      /*
       * The following is the real annotated information
       */
      ctf_string(key, key)
      ctf_string(val, val)
      )
  )
```

3. **trace-name** refers to member variable name of object trace of struct blkin_trace

4. **endpoint-name, endpoint-port, endpoint-ip** refer to
Directory: blkin-lib/zipkin_c.h

```
struct blkin_endpoint {
            const char *ip;
            int16_t port;
            const char *name;
}
```

5. **trace-info.trace_id, trace-info.span_id, trace-info.parent_span_id**, refer to trace object of trace class in Ztracer namespace.
Directory: blkin-lib/ztracer.hpp

```
class Trace : private blkin_trace {
   private:
string _name; // storage for blkin_trace.name, see copy_name()

   public:
// default constructor zero-initializes blkin_trace valid()
// will return false on a default-constructed Trace until
// init()
Trace()
```

```
 {
// zero-initialize so valid() returns false
name = NULL;
info.trace_id = 0;
info.span_id = 0;
info.parent_span_id = 0;
endpoint = NULL;
}

}
```

6. **annotation-key and annotation-val_str** refer to annotation object of the following structure.

Directory: blkin-lib/zipkin_c.c

```
    struct blkin_annotation {
 blkin_annotation_type type;
 const char *key;
 union {
const char *val_str;
int64_t val_int;
 };
 const struct blkin_endpoint *endpoint;
    };
```

II. If annotation-type == ANNOT_INTEGER We call the tracepoint() function with similar arguments, except a few changes.
1. **keyval_integer**

Directory: blkin-lib/zipkin_trace.h

```
TRACEPOINT_EVENT(
        zipkin,
        keyval_integer,
        TP_ARGS(const char *, trace_name, const char *, service,
                int, port, const char *, ip, long, trace,
                long, span, long, parent_span,
                const char *, key, int64_t, val ),

        TP_FIELDS(
                /*
                 * Each span has a name mentioned on it in the UI
                 * This is the trace name
                 */
                ctf_string(trace_name, trace_name)
                /*
```

```
                 * Each trace takes place in a specific machine-endpoint
                 * This is identified by a name, a port number and an ip
                 */
                ctf_string(service_name, service)
                ctf_integer(int, port_no, port)
                ctf_string(ip, ip)
                /*
                 * According to the tracing semantics each trace should have
                 * a trace id, a span id and a parent span id
                 */
                ctf_integer(long, trace_id, trace)
                ctf_integer(long, span_id, span)
                ctf_integer(long, parent_span_id, parent_span)
                /*
                 * The following is the real annotated information
                 */
                ctf_string(key, key)
                ctf_integer(int64_t, val, val)
                )
        )
```

2. **annotation-val_int**
Directory: blkin-lib/zipkin_c.c

```
struct blkin_annotation {
    blkin_annotation_type type;
    const char *key;
    union {
   const char *val_str;
   int64_t val_int;
    };
    const struct blkin_endpoint *endpoint;
};
```

III. Or else
It calls tracepoint(), with a difference in argument.
1. **timestamp**
Directory: blkin-lib/zipkin_trace.h

```
TRACEPOINT_EVENT(
        zipkin,
        timestamp,
        TP_ARGS(const char *, trace_name, const char *, service,
                int, port, const char *, ip, long, trace,
                long, span, long, parent_span,
                const char *, event),
```

```
TP_FIELDS(
        ctf_string(trace_name, trace_name)
        ctf_string(service_name, service)
        ctf_integer(int, port_no, port)
        ctf_string(ip, ip)
        ctf_integer(long, trace_id, trace)
        ctf_integer(long, span_id, span)
        ctf_integer(long, parent_span_id, parent_span)
        ctf_string(event, event)
        )
)
```

5. Scratch Notes on Equivalence and Current work

1. The inject() function in Jaeger is similar to the event() function in Blkin. Inject() accepts an argument ctx which corresponds to the trace object of Blkin. The inject() function injects SpanContext into a carrier.

```
opentracing::expected<void>
Inject(const opentracing::SpanContext& ctx,
       const opentracing::HTTPHeadersWriter& writer) const override
{
    const auto* jaegerCtx = dynamic_cast<const SpanContext*>(&ctx) ,;
    if (!jaegerCtx) {
        return opentracing::make_expected_from_error<void>(
            opentracing::invalid_span_context_error);
    }
    _textPropagator.inject(*jaegerCtx, writer);
    return opentracing::make_expected();
}
```

2. SpanContext() in Jaeger does what init() function in Blkin does

3. We should to look at
   1. The SpanContext() concept and focus on Baggage itemswhich are key value pairs 2. Opentracing tutorials(Python) all of them to understand the concept

References

[1] https://ceph.com
[2] https://lttng.org