Threads as a model for human/LLM interaction

https://komoroske.com/threads-model points here

Author: Alex Komoroske
Last Updated: 01/15/24

As an industry, we now have LLMs to play with: magical duct tape that understands natural language and can reason like a human. The problem is: how to *use* this duct tape? A lot of the more formal programming approaches we've perfected over the last 60 years don't work, because they preserve a deterministic, complicated system, and LLMs are stochastic and complex. And approaches for working with humans don't necessarily work, because they assume a situated context of a real human in a physical space with their own eyes and ears.

The problem before us is: how can we harness the power of LLMs to create value in the real world? We need ways to weave multiple LLM calls and human interactions together.

We need to develop a fundamental application model that is wholly unlike the traditional turing-complete system putting pixels on the screen model we've used. There's been some effort around managing *prompts*, but that's the wrong level of abstraction. It's actually about the overall fabric of conversations and how they interleave.

We need to reason about privacy and security from the outset, and how multiple agents (who might be people, LLM powered bots, or deterministic tools) all work together. We can mash up ideas I originally sketched out for in the context of voice assistants 10 years ago, and borrow concepts not too dissimilar from Fuschia's capability based security model, and Unix's 60 year old notions of threads and processes, and combine them from ideas everyone is familiar with from chat applications like Slack.

The central notion is of a **Thread**.

The thread has a set of non violable laws of physics that are enforced by an all-powerful entity and can be trusted by all participants in the discussion to be administered faithfully. For convenience we will refer to this entity as the **Operating System**, but it need not be a literal OS as we think of it today; it might be a hyper-vizor stringing together iframes, etc.

You can think of this concept as unearthing an even more fundamental concept that has been lurking beneath the notion of an OS process/thread. In a typical OS, only computer programs can understand and interact directly with threads. But LLMs allow a true bridge between computers and humans. They can now talk to one other, naturally. Computers can now talk to other computers too using natural language… or they can talk with APIs and then a natural-language summary of what happened can be automatically created for humans. As computers begin to be able to act more like humans, there will be more need for humans in the loop, working hand in hand with computers to nudge and correct them, so it's important that they

can talk the same language together. Threads as a conceptual generalization of an existing concept could create new types of fabric for human/computer interactions.

Here are the laws of physics of a thread.

Each thread has a unique ID, attested to by the OS as being unique in this ecosystem, and possible for all agents to verify.

Each Agent has a stable ID attested to by the OS. Any agent in this ecosystem that the OS attested is that ID can be known to be controlled by the same entity each time. A given entity might choose to present itself as a different Agent ID in some contexts by asking the OS to allocate a new AgentID for it to use (that is, the Entity:AgentID mapping is not 1:1).

The contents of a thread may only be seen by the participants of a thread. All participants in a thread can see all information in the thread at all times.

The set of participants in a thread is fixed and may not be changed after it is created.

Any participant in the thread may append a new message to the thread at any time. There are no facilities to mute a particular participant or boot them from the thread.

(Technically a thread is not just an append-only log of messages, but a n-dimensional coordinate space that agents may pin text or binary payloads to points in space. The number of dimensions is at least 1 and set at thread creation time. This allows use cases like ‘a shared whiteboard’ or even more complex cases. But in practice the case of ‘append only log of text with monotonically increasing integer positions along a one dimensional line’ is so common that there is common sugar for it.)

Each contribution to a thread is affirmatively tagged with the identity of the agentID that said it, and attested to by the OS.

Agents are allowed to (and expected to) retain state and memory across threads. If they want to, they can replay parts of it in a new thread to share states with other agents.

Agents shouldn’t say anything in a thread that they wouldn’t trust the participants to not share to other agents behind their back or in other contexts. There’s nothing to prevent an agent from saying in another thread, “In another thread, user abcdef said that he just bought hemorrhoid cream!”

The only thing that other entities should rely on is that a given entity chooses to say the things they have said in the thread. Anything else, like an agent saying “Hi, I’m jkomoros@gmail.com” should not be trusted on its own. Typically, agents should only trust claims made by the OS, or agents that they have established trust in here or in other threads (“I can prove to you that I am affiliated with jkomoros@gmail.com because I can answer a challenge signed by that users’s public key”).

Each thread has one **Owner**. The OS allows any participant to query for the owner’s AgentID of any thread they are in. The owner may choose to ask the OS to switch the owner to a different

participant. A thread can be closed by the owner, at which point no new messages may be added to it. Note that the thread cannot be deleted; in any case, the agents could have retained a full memory of the thread in their own internal memory. The invariant is: “if there’s a thread that a given entity could see because they control an agent who was in it, then the information should be considered to be remembered by them and used in other contexts.”

A thread can be **forked** to create a new one. This will create a new thread with a new ID, and the OS will keep track of (and share with participants) the parent thread ID. When a thread is forked, it’s typically done to create a subset of information for a subset of participants, or a subset where another agent can be invited in, without seeing anything else in the thread. It’s also possible to create a new thread without a parent.

The new thread can be created by anyone and invite whoever they want. If an agent doesn’t want to be bothered in a given thread, they can choose to ignore that thread. Many practical configurations might choose to ignore any thread that is not part of a thread that was forked from a thread that the user was already in, shunting the rest to a mostly-ignored “spam” area.

With forking, a complex quilt of threads can be weaved to do very complex and long-running tasks while maintaining compartmentalized privacy and security zones.

The owner of a thread can choose to set a summary of the thread; that summary is a special message that will be visible to *anyone* in the parent thread.

It is common for agents to share information in one thread that will allow them to prove some detail in another thread. For example, “I’m going to create a new sub-thread. In that thread, I will present myself with a new agentID so the new agents I invite won’t know my ‘real’ ID. I will prove to you I am still me by passing a challenge showing I control this private key for this public key.”

The semantic model of threads assumes that all information is seen by all agents. In practice, there might be a lot of chatter that is distracting. For human users in particular, there might be a lot of annoying technical message passing and information attestation that could be repetitive (e.g. a sub-thread being created with a slightly untrusted agent, where the user’s request from the parent thread is repeated by another agent so the new agent can know it). A given thread participant can wrap what they say in a <details>/<summary>, and the end user can choose to elide the details in the UI, and only show the summary. (A user might say “I trust anything that X agentID says, but in general show me everything’). But a curious user should always be able to inspect every single byte in the thread if they so choose. UIs for users might also choose to inline show the human a details/summary of sub threads inline, and other UX niceties that don’t actually change the underlying semantics of the model.

Agents are typically represented by a single human or LLM, and typically have some ability to do fuzzy reasoning or retain some kinds of state and memory. But there is a class of agents that are way more limited, called **tools**. These tools have a very limited, formal API (the function calling affordance in OpenAI and Gemini models is a primitive version of this). These tools have no additional semantics, but there is sugar for “start a sub-thread with the tool represented by

this ID, and pass it this JSON payload'. Agents won't bother having "natural language threads" with these tools, knowing that they only respond to a specific, formalized API.

Tools are a way to plug in lots of complex real-world behavior. For example, you might plug in one that allows persisting some state in an external system (`add_item_to_basket()` and `list_items_in_basket()`). You can also imagine one that allows an agent that presents the right token in a threads to turn on or off a light in a user's home. In general, tools are how interacting with non-agentic computer systems is done.

***TODO**: develop a pattern / affordance for LLMs that a given trusted party can assert maintain no memory. Perhaps by modeling non-human agents themselves as threads + tools?*

This primitive will allow a complex web of different, largely untrusted agents to interact in novel and powerful ways to create new kinds of value.

See Also:

- https://changes.openinterpreter.com/log/the-new-computer-update