

RTT, messages et synchronisation

Chaîne de synchronisation

BrainCloud RTT permet au serveur d'avertir les clients déjà connectés lorsqu'un document ou un état métier change. Côté serveur, la couche commune est `_utils/utils_message.js`. Côté Unity, `BackEndBrainCloud` reçoit les messages et les distribue au runtime.

```
sequenceDiagram
    participant S as Cloud Script
    participant M as utils_message
    participant RTT as BrainCloud RTT
    participant B as BackEndBrainCloud
    participant D as MessagesDispatcher
    participant C as Controller
    S->>M: sendMessageToTeam / User / Sim
    M->>RTT: message type + data
    RTT-->>B: OnRTTMessageReceived
    B->>D: ServerMessage
    D-->>C: listener
    C->>C: refresh du modèle / UI
```

Helpers serveur

`utils_message.js` expose notamment `sendMessageToTeam`, `sendMessageToMemberList`, `sendMessageToSim`, `sendMessageToActStudents`, `sendMessageToUser`, `sendMessageToTeamAndTeacher` et des messages spécialisés comme `sendLedgerPageChangedMessage`.

Connexion RTT

Après `AuthenticateUniversal`, `BackEndBrainCloud.EnableRTT()` appelle `BrainCloud.RTTService.EnableRTT`. Sur succès, le client enregistre `RegisterRTTMessagingCallback(OnRTTMessageReceived)`. Si la socket est fermée, le code tente de réactiver RTT; si l'authentification n'est plus valide, il réutilise la fonction d'authentification.

Règle pour une nouvelle feature

Une écriture serveur réussie ne suffit pas si plusieurs utilisateurs voient le même état. Après la sauvegarde, déterminer qui doit être averti : utilisateur, équipe, équipe + professeur, activité ou simulation. Le listener Unity doit ensuite mettre à jour le modèle local avant l'UI.

Diagnostic

1. Vérifier que le Cloud Script envoie bien un message.
2. Vérifier le type et la payload.
3. Vérifier que RTT est connecté.
4. Trouver le `MessageType` côté Unity.
5. Trouver le listener dans le controller.
6. Vérifier si le listener applique la donnée reçue ou redemande un document.

Ajouter un nouveau message RTT de A à Z

1. Définir le message côté Unity

Ajouter la constante dans `MessageTypeBase.cs` si le message est commun, ou `MessageTypeByModule.cs` s'il appartient au framework By Module. La string doit être exactement la même que celle envoyée par le serveur.

2. Envoyer depuis le serveur après la sauvegarde

Utiliser l'helper d'audience approprié dans `utils_message.js` :

- `sendMessageToUser` ;
- `sendMessageToTeam` ;
- `sendMessageToTeamAndTeacher` ;
- `sendMessageToSim` ;
- `sendMessageToActStudents` .

```
saveDocumentData("teams", teamData.uid);

sendMessageToTeamAndTeacher(teamData, "my_data_changed", {
    teamUid: teamData.uid,
    value: teamData.myValue
});
```

3. Enregistrer le listener dans le controller

```
public override void SetListeners()
{
    base.SetListeners();
    Game.MessagesDispatcher.RegisterListener(
        MessageTypeBase.MY_DATA_CHANGED,
        OnMyDataChanged
    );
}

public override void RemoveListeners()
{
    base.RemoveListeners();
    Game.MessagesDispatcher.UnRegisterListener(
        MessageTypeBase.MY_DATA_CHANGED,
        OnMyDataChanged
    );
}
```

4. Traiter la payload

Le handler vérifie l'UID/context, applique les données au modèle puis déclenche l'événement UI nécessaire. Ne pas recharger tout le document si quelques champs suffisent.

5. Grosse payload : ne pas l'envoyer en RTT

Pour des résultats/module data volumineux, envoyer une instruction de récupération. Le code By Module utilise par exemple `execute_request` pour demander au client de récupérer les résultats plutôt que d'envoyer tout le résultat dans le message.

Ordonnancement réel côté client

`BackEndBrainCloud.OnRTTMessageReceived()` transforme la payload en `ServerMessage`.

`MessagesDispatcher` met les messages dans une queue, attend brièvement, les trie par timestamp puis appelle chaque listener enregistré. Ce buffer réduit les problèmes d'ordre entre messages proches.

Le cas `get_data`

`OnRTTMessageReceived()` reconnaît les messages qui demandent un fetch supplémentaire. Dans ce cas le client charge la donnée avant de construire le `ServerMessage` final. Utiliser ce pattern lorsque la source de vérité doit rester le document serveur.

Test minimal multi-client

1. Ouvrir deux clients sur la même équipe/simulation.
2. Faire l'action sur A.
3. Vérifier que le document est sauvegardé.
4. Vérifier le log RTT de B.
5. Vérifier que le bon listener B est exécuté une seule fois.
6. Changer de simulation sur B puis refaire l'action : aucun ancien controller ne doit encore recevoir le message.

Revision #2

Created 2026-09-18 19:10:03 UTC by Mathis

Updated 2026-09-18 20:08:00 UTC by Mathis